2

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A RESOURCE CONSTRAINED LOOP PIPELINING TECHNIQUE FOR PERFECTLY-NESTED LOOP STRUCTURES

by

Thor Davis Aakre

September 1993

Thesis Advisor: Dr. Amr M. Zaky

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE September 1993	3. REPORT TYPE AND DATES COVERED Master's Thesis, July 1991 - September 1993

4. TITLE AND SUBTITLE A Resource Constrained Loop Pipelining Technique For Perfectly-Nested Loop Structures (U)	5. FUNDING NUMBERS
6. AUTHOR(S) Aakre, Thor Davis	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-500	10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This thesis presents a new technique for loop pipelining of perfectly-nested for-loop structures which is designed to optimize loop execution on VLIW machines. Previously implemented loop pipelining techniques provide limited performance benefit because they explicitly include the constraints imposed by a loop's cyclic dependences in their loop pipelining process. Some loop pipelining techniques have also ignored the realistic constraint of finite resource availability in the creation of final pipelined execution schedules.

The new approach presented in this thesis eliminates the problem of cyclic dependences by first applying a linear transformation to the nested loop index space to ensure a cycle-free innermost loop, which is then pipelined using modulo scheduling for a known set of resources. The transformation guarantees that the target machine's available resources are the only limit to the amount of exploitable fine-grained parallelism within the innermost loop. This results in pipelined execution schedules having near-optimal Inter-Iteration Initiation Intervals (IIII) with the achievable performance being scalable with the addition of resources. Consequently, our loop pipelining method utilizes more fine-grained parallelism than other loop pipelining techniques which directly incorporate a loop's cyclic dependences in their pipelining process. We also explicitly provide a procedure for creating the resultant pipelined execution schedules. In addition, we investigate the negative effect that the transformation has on data locality and the cache miss rate, as well as the use of iteration space tiling to restore data locality and cache miss rate to the levels expected from sequential loop execution.

14. SUBJECT TERMS Loop Pipelining, Inter-Iteration Initiation Interval, Modulo Scheduling, Data Dependence Graph, Unimodular Transformation, Modulo Variable Expansion	15. NUMBER OF PAGES 181
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT Unlimited

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

A RESOURCE CONSTRAINED LOOP PIPELINING TECHNIQUE FOR PERFECTLY-NESTED LOOP STRUCTURES

by

Thor Davis Aakre
Lieutenant, United States Navy
B.A., St. Olaf College, 1984

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

September 1993

Author: Thor Davis Aakre
Thor Davis Aakre

Approved By: Amr Zaky
Amr M. Zaky, Thesis Advisor

[illegible]
Man-Tak Shing, Second Reader

[illegible]
Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

This thesis presents a new technique for loop pipelining of perfectly-nested for-loop structures which is designed to optimize loop execution on VLIW machines. Previously implemented loop pipelining techniques provide limited performance benefit because they explicitly include the constraints imposed by a loop's cyclic dependences in their loop pipelining process. Some loop pipelining techniques have also ignored the realistic constraint of finite resource availability in the creation of final pipelined execution schedules.

The new approach presented in this thesis eliminates the problem of cyclic dependences by first applying a linear transformation to the nested loop index space to ensure a cycle-free innermost loop, which is then pipelined using modulo scheduling for a known set of resources. The transformation guarantees that the target machine's available resources are the only limit to the amount of exploitable fine-grained parallelism within the innermost loop. This results in pipelined execution schedules having near-optimal Inter-Iteration Initiation Intervals (IIII) with the achievable performance being scalable with the addition of resources. Consequently, our loop pipelining method utilizes more fine-grained parallelism than other loop pipelining techniques which directly incorporate a loop's cyclic dependences in their pipelining process. We also explicitly provide a procedure for creating the resultant pipelined execution schedules. In addition, we investigate the negative effect that the transformation has on data locality and the cache miss rate, as well as the use of iteration space tiling to restore data locality and cache miss rate to the levels expected from sequential loop execution.

ACKNOWLEDGEMENTS

I would like to thank Dr. Amr M. Zaky, whose initial interest in the subject of loop pipelining was the foundation on which this thesis was produced. His continued support, enthusiasm, patience, and guidance were invaluable assets for the completion of this work.

I also would like to thank my wife, Anne, for doing more than her share of life's normal routines, for supporting my schedule, and for providing alternative activities to balance the load.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

With the ever increasing demand for higher performance in computer processing, constant research is being conducted in an attempt to find methods to execute program instructions faster. One area of this research emphasizes the use of concurrent processing to exploit the independent components of a program by processing these components in parallel. The level at which this parallelism is exploited can vary from a coarse-grained parallelism (e.g., from fully independent processes and procedures, independent loop iterations, etc.) to fine-grained parallelism (e.g., from independent machine instructions or microinstructions).

While coarse grained parallelism may be the simplest for which to plan and even to design programming tools to identify and exploit, for many applications, it does not often provide enough parallelism to fully utilize the resources made available for concurrent use. By considering finer grained components, such as instructions or micro-instructions, a greater number of independences should be uncovered. As a result, exploiting the parallelism at this level provides a better chance of keeping resources busy. The problem, however, is finding an effective and efficient method of identifying and harnessing the fine-grained parallelism in existing code to create execution schedules which maintain the original program semantics. Of particular focus in determining a solution to this problem is the handling of the fine-grained parallelism present in loop structures, which consumes a large percentage of the total execution time of scientific applications.

Several general machine types have been proposed which attempt to exploit the fine-grained parallelism in programs, two of which are the *Superscalar Machines* and the *Very Long Instruction Word* (VLIW) *Machines*. In Superscalar Machines, several instructions in a sequence are considered for concurrent execution. Dependences between, as well as other characteristics associated with, these instructions are examined and, based on the results, a subset of the instructions are issued to multiple functional units for parallel execution. The real limit to the effectiveness of the Superscalar architecture involves the run time overhead required to dynamically determine the inter-dependence within a sequence of instructions. To minimize this overhead, only a limited number of instructions can be considered for execution at any one time. Techniques which utilize compile time analysis of the program code could help eliminate or at least ease this run time overhead.

This is the approach used for VLIW machines, which require compile time evaluation of inter-instruction dependences, followed by the combination of individual independent instructions into one long instruction word. The individual independent instructions packaged into the long instruction are then fetched as one instruction, and simultaneously issued to multiple function units (Figure 1). This allows a more effective analysis of dependences without affecting run time performance, and results in better utilization of fine-grained parallelism.

One technique specifically tailored for use with VLIW machines is called *Trace Scheduling* (Fisher, [Ref. 1]). Trace Scheduling first requires a selection (called *trace selection*) of the most likely trace through the code. Loop unrolling is used to create long traces, but requires the assumption that certain loop control conditionals are taken with a high probability. A second step, *trace compaction*, is then used to analyze the trace for dependences and compress the code into the VLIW format. Only the most probable traces are scheduled this way. Correction code is required for those cases in which the path of execution veers off of the selected trace path. In addition to the complexity of this method, Lam [Ref. 2] notes that there is the possibility of exponential code explosion. Another deficiency of trace scheduling, as noted by Zaky and Sadayappan [Ref. 3], is that there is no easy way to determine how much unrolling in any specific circumstance would produce better utilization of resources and better performance. Therefore, the resultant ad-hoc methods of loop unrolling that are often used to determine the needed amount of unfolding are not effective ways to best create VLIW instruction schedules.

Code for determining the value of
C=(3*A)+(4*B):

```
S1:    LD R0, A
S2:    MULTI R1, R0, #3
S3:    LD R2, B
S4:    MULTI R3, R2, #4
S5:    ADD R4, R1, R3
S6:    ST C, R4
```

Sequentially executed code would take six time units to execute if each instruction required one time unit to execute

With a VLIW machine with two fully capable processors, the VLIW instruction is comprised of two sub-instructions, one for each processor. No-Op instructions are executed when no specific sub-instruction is assigned. Evaluation of the above code at compile time would determine that the code is executable in four time units

Processor

time	P1	P2
1	LD R0, A	LD R2, B
2	MULTI R1, R0, #3	MULTI R3, R2, #4
3	ADD R4, R1, R3	
4	ST C, R4	

Four individual VLIW instructions, each with a sub-instruction assignment to a separate processor in the target machine.

Figure 1: Translation of Sequential Code into VLIW Instructions

An alternative to trace scheduling is *Loop Pipelining* (or Software Pipelining). Loop Pipelining is a technique whereby instructions from different loop iterations are interleaved without unrolling the loop. The interleaving allows exploitation of fine-grained parallelism between instructions of different iterations by combining these independent instructions into a single long instruction. A restructured loop body of VLIW instructions is created and replaces the original loop.

The main idea behind the technique is to generate a compact loop body of VLIW instructions which maintain the semantics of the original loop structure. For example, consider the *Data Dependence Graph* (DDG) in Figure 2 for a single loop. In the DDG, each node represents an instruction with arcs representing data dependences between the instructions. The labels are in the form (latency)/(loop delay). The latency refers to the number of time units required between the start of one instruction and the start of the dependent instructions. The loop delay identifies the relationship between the iteration of the dependent instruction as compared to the iteration of the instruction on which it depends.

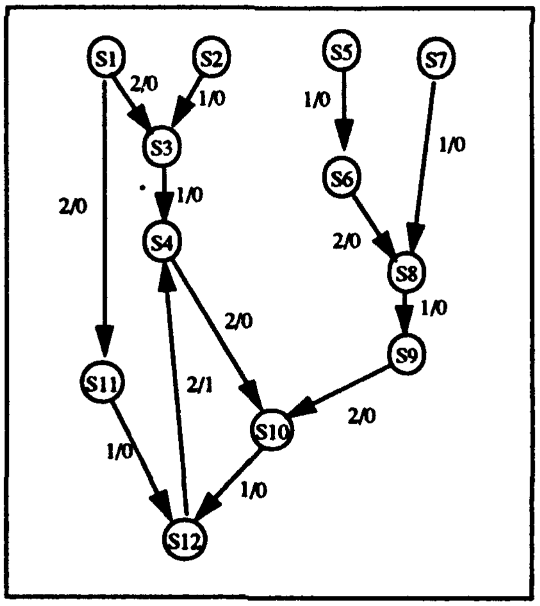


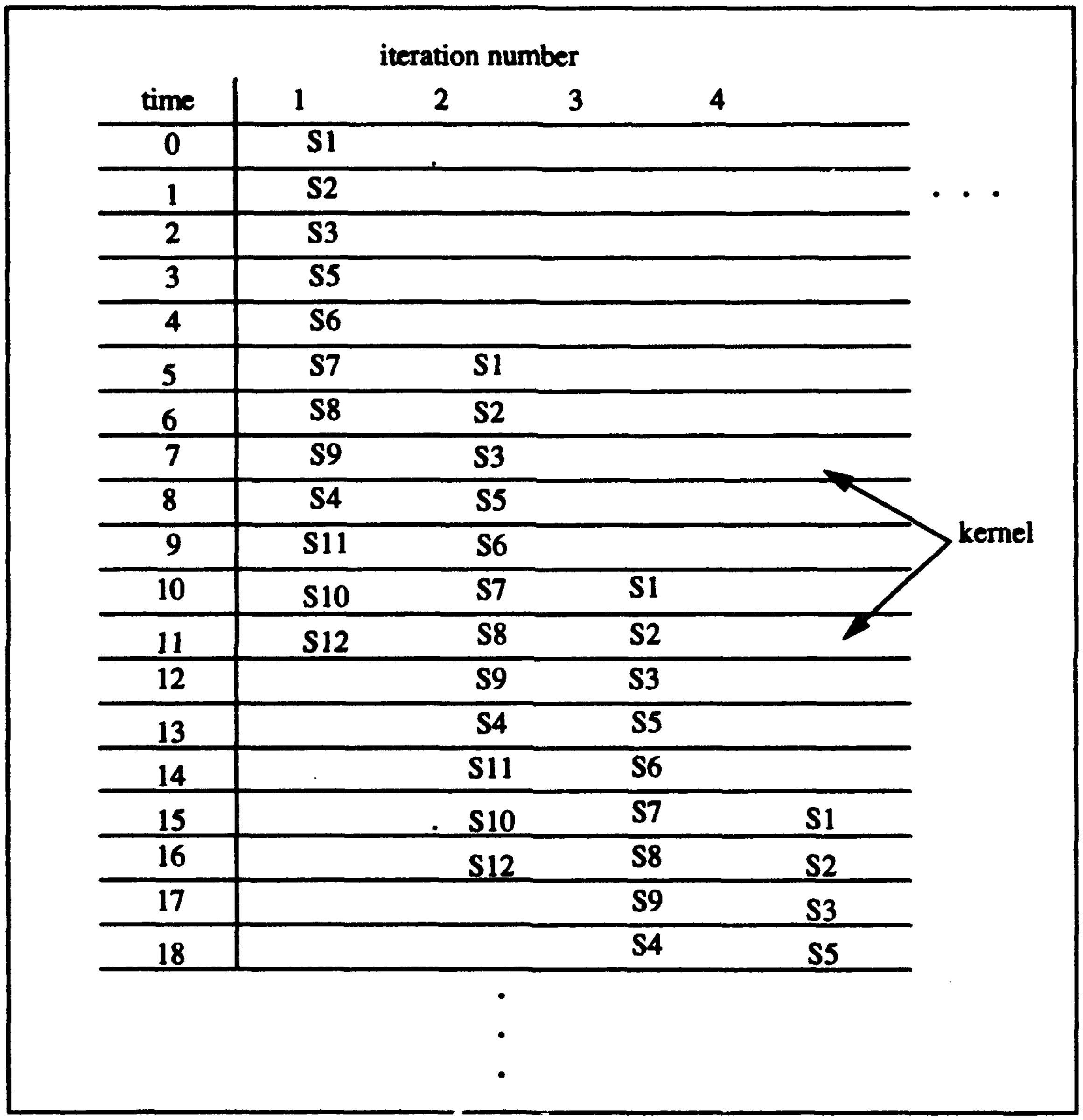
Figure 2: Data Dependency Graph

In this example, assume that there are three processing elements available and that any of the processing elements can execute any of the instructions in one unit of time. The iterations can then be scheduled as in Figure 3, with instruction from different iterations overlapping in time, but with no more than three instructions being executed at any one time (because there are only three available resources). The VLIW instructions which are created are comprised of the sub-instructions which are executed at the same time in the schedule.

As can be seen by this schedule, a recurring pattern develops in which a new iteration is started every five time units, even though it takes twelve time units to complete any one iteration. This is the pipelining effect. The recurring pattern which first occurs at time 7 thorough 11, is referred to as the *kernel* of the new schedule. The kernel executes any instruction of the loop body only once, although the instructions in any kernel may come from different iterations.

To take advantage of the multiple resources available, the original loop can be restructured to include the kernel pattern as the new loop body, which executes the twelve instructions in five time periods. The amount of time needed to execute the kernel is also the time between subsequent starting of iterations. This time is labeled the *Inter-Iteration Initiation Interval* (IIII). The IIII becomes a measure of the throughput of the system and of how well the resources are being utilized. The smaller the IIII, the greater completion rate of the loop iterations and the better the resources are being used. It is obvious that any software pipelining method must have as its goal the creation of a kernel with the minimal IIII.

The *Modulo Scheduling* technique developed by Rau and Glaeser [Ref. 4] was shown to be able to schedule acyclic DDG's to create a loop body kernel which takes full advantage of the available resources, and therefore yields a minimal IIII for the given set of resources. In many loop structures, loop carried dependences exist between the iterations of the loop body. Although their existence are not a sufficient condition for creating cyclic dependences, they are necessary, and often create cyclic dependences which are displayed



time	1	2	3	4
0	S1			
1	S2			
2	S3			
3	S5			
4	S6			
5	S7	S1		
6	S8	S2		
7	S9	S3		
8	S4	S5		
9	S11	S6		
10	S10	S7	S1	
11	S12	S8	S2	
12		S9	S3	
13		S4	S5	
14		S11	S6	
15		S10	S7	S1
16		S12	S8	S2
17			S9	S3
18			S4	S5

iteration number
. . .
kernel
.
.
.

Figure 3: Timing Schedule for Iterations Represented by DDG of Figure 2

as cycles in the DDG for the loop. Data dependence cycles in a DDG introduce additional constraints on the minimum length of IIII. As a result, cycles can limit the size of the kernel schedules which can be produced and restrict the performance benefit which can be obtained by loop pipelining.

Modulo Scheduling methods presented for pipelining single loops with cyclic data dependences are described by Aiken and Nicolau [Ref. 5], Lam [Ref. 2], Rau, Schlansker and Tirumalai [Ref. 6], and Zaky [Ref. 7]. However, these methods directly incorporate the constraints caused by the cyclic dependences into the scheduling procedure. As noted, this restricts the minimum size of the IIII and prevents the methods from fully benefitting from extra resources.

Because the time spent executing perfectly-nested loop structures can dominate program runtime, previous loop pipelining techniques must be expanded to incorporate these structures. Loop unrolling can be applied along multiple dimensions in an attempt to eliminate dependence cycles and expose additional fine-grained parallelism beyond that available from single dimension unrolling. This is the intent behind the *Loop Quantization* method described by Nicolau [Ref. 8]; however, just as with trace scheduling, determining the amount and the direction of unrolling required to guarantee good results is not easy, and the benefit may not justify the complexity of the effort.

Alternatively, the modulo scheduling techniques presented by Zaky [Ref. 7] and Kim and Nicolau [Ref. 9] identify significant fine-grained parallelism across the entire iteration space of a nested loop. Both determine, via linear timing functions, the sequential starting times of sets of independent instructions which can be executed in parallel. However, neither provides a concrete solution for mapping the instructions on finite resources.

In summary, previously presented techniques for loop pipelining have either been inherently limited by the existence of cyclic dependences, have applied ad-hoc methods in hopes of improved performance, or have ignored the realistic considerations for resource constraints, execution schedule production, and actual creation of final code products.

These failures were the motivation behind the development of the loop pipelining technique described in this thesis.

The technique developed for this thesis emphasizes the efficient use of available resources. It combines a method for identifying sets of independent iterations in multi-dimensional space with a loop pipelining technique based on Modulo Scheduling of acyclic DDG's mentioned earlier. The result is a simple procedure yielding useful execution schedules with near-optimal IIII. The advantage over previously proposed perfectly-nested loop pipelining methods is its simplicity and the exploitation of fine-grained parallelism to the extent allowed by available resources. In addition, a code generation procedure is provided for producing the final code structure using the pipeline schedule resulting from the application of the technique.

Chapter II of this thesis first describes, in more detail, the Modulo Scheduling technique for acyclic DDG's. It then highlights the difficulties encountered when attempting to apply a general Modulo Scheduling technique directly to cyclic DDG's, as well as the application of the technique to perfectly-nested loop structures.

Chapter III describes the proposed loop pipelining technique which can be used to create software pipelined schedules for n-dimensional perfectly-nested loops. The chapter first details the loop transformation process, which converts the original loop structure into one in which the inner loop can be pipelined using the Modulo Schedule method for acyclic DDG's. The chapter then outlines the process for creating the loop pipelined schedule via the Modulo Scheduling method.

Chapter IV explains the process of code generation using the loop pipelining technique presented. In particular, it modifies the technique to include the scheduling of loop control instructions. In addition, it provides a summary of the special machine hardware support requirements that are assumed to be true for the code generation process. Several code generation considerations are addressed, and a schematic diagram is presented to aid in summarizing the require code segments which must be included in the final loop structure created. Lastly, the algorithm of the code generation is presented.

Chapter V summarizes the performance benefits of the proposed loop pipelining technique and analyzes the complexity of the code generation process.

Chapter VI addresses the additional concern of data locality, particularly in light of the negative effects the loop pipelining technique might create, and the possible solutions to minimize these negative effects.

Chapter VII presents are review of and the conclusions to the work conducted, as well as identifies the necessary extensions of the research required to fully explore and implement the technique presented.

II. BACKGROUND

The Modulo Scheduling technique described by Rau and Glaeser [Ref. 4] has been used to loop pipelined loop structures which are represented by both cyclic and acyclic DDGs. In this chapter, the specifics of the Modulo Scheduling technique are described in more detail for both of these applications. The concern of this thesis, however, is the application of the scheduling technique to perfectly-nested loop structures, which is addressed at the end of the chapter. The basic modulo scheduling methods described below were presented in detail by Rau and Glaeser [Ref. 4], and are used as a general basis for all modulo methods subsequently developed.

A. MODULO SCHEDULING OF ACYCLIC DDGs

For loops with no cyclic dependences, Modulo Scheduling methods can create pipelined schedules which utilize resources to the maximum benefit. The method accomplishes this by creating a pipelined kernel schedule which has the *smallest IIII* possible under the circumstances and constraints imposed by the specific resources made available. The technique first determines the minimum IIII possible, and then applies scheduling methods to create the pipelined execution schedule which will become the new pipelined kernel.

1. Determining The Inter-Iteration Initiation Interval

The first step in applying a modulo scheduling technique to acyclic DDGs is the determination of the IIII. This is done by examining the instructions in the loop body and compares the resource requirements for executing the instructions with the resources available in the VLIW machine. The IIII which is chosen for the loop pipelined schedule is that IIII which satisfies the needs of the most limiting resource type. That is, there must be enough instruction slots available in the kernel of the pipelined schedule to ensure that all instructions can be fit into the schedule.

The calculation for the IIII is found by the equation:

$$IIII_{Lowerbound} = max_{r \in R} \left\lceil \frac{Total\ Time\ For\ r}{N_r} \right\rceil \qquad \text{(Eq. 1)}$$

where R is the set of all resource types, with r being one type of resource.
Total Time For r is the total amount of time that the resource type r is required to be used the instructions
N_r is the number of resource units of type r

It is important to note that the "Total Time For r" required of a resource type is not dependent upon the latency values of instructions as shown in a DDG. Rather it is dependent upon the delay of a functional unit when executing an instruction. This ***resource delay*** is the number of time units following the start of one instruction during which the resource is unable to start another instruction. As a result the value of "Total Time For r" in the above equation is really $\Sigma_{i \text{ using } r}(resource\ delay_r)$. This is a function of the resources' pipelining capability. As an example, consider the DDG shown in Figure 4, which is a modification of Figure 2, with cyclic dependences removed.

In Figure 4, S3 cannot start until at least one time unit after the start of S2 due to instruction dependence. If we assume that S2 utilizes an adder to produce a value that is used by S3, then the latency of "1" means that the value produced by S2 is not available to S3 until one time unit from the start of the S2. If we assume that S3 requires use of the same adder as S2 and that the adder can only start a new instruction every two time periods, then the adder's resource delay is two time units. This prevents S3 from executing for two time units after the start of S2 (see Figure 5.a).

If another adding unit is used to execute S3, then S3 would not be affected by the resource delay and could start one time unit later than S2 (see Figure 5.b).

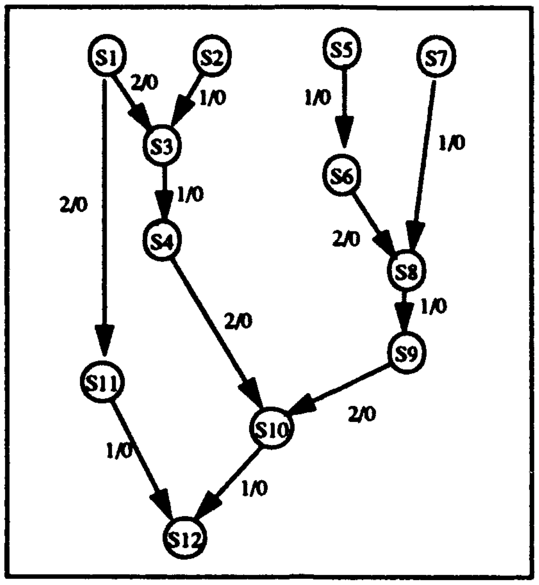


Figure 4: Acyclic Data Dependency Graph

time	adder 1
1	S2
2	
3	S3

a. With One Adder

time	adder 1	adder 2
1	S2	
2		S3

b. With Two Adders

Figure 5: Scheduling of S2 and S3 From Figure 4 With an Adder Resource Delay of Two Time Units, With One and Two Adders Available

It is important to note that IIII calculation is independent of the graph structure and depends only on the nodes. That is, there is no input to the calculation of the IIII involving the latency or loop delays of each of the arcs. The type of instructions represented by the nodes and the resources available are the only required inputs for calculating the IIII.

To illustrate the calculation of the IIII, consider again the example in Figure 4. Assume that the resource delay is one time unit for all instruction types, that the machine for which the example is created has two adders, a multiplier, and a load/store unit. Also, assume that the DDG nodes S2, S3, S5, S7, S8, S10, and S11 are adder instructions, instructions S1 and S6 are multiplier instructions, and instructions S4, S9, and S12 are load store instructions. Then the calculation of the IIII becomes:

$$IIII_{Lowerbound} = max\left\lceil \frac{7}{2}, \frac{2}{1}, \frac{3}{1} \right\rceil = 4 \qquad \text{(Eq. 2)}$$

2. Creating The Modulo Resource Reservation Table

Once the IIII has been determined, the next step in applying Modulo Scheduling is to create a *Modulo Resource Reservation Table* to aid in scheduling the DDG instructions. The Modulo Resource Reservation Table identifies the relative starting times of instruction nodes in the kernel. The intent is to assign instructions to the table in a way that minimizes the IIII ultimately produced. The assignment of instructions to the table slots is purely an exercise in bin packing. That is, the instructions are assigned to the proper resource while maintaining the resource delay requirements. If the resource delays for the instruction nodes are all one time unit, the instructions can be placed randomly in a table and meet the resource delay requirements using the lower bound IIII.[1] If some resource

1. All mappings of instructions to resources in the Modulo Resource Reservation Table when the resource delay is one yield the same IIII. However, different mappings affect the number of different iterations which are represented by instructions in the kernel. This creates different characteristics in the transition which is needed before the pipelined schedule is used, as will be discussed later.

delays are more than one unit, then the lower bound IIII may not be adequate, requiring that the final IIII be determined using some bin-packing technique.

For example, consider the simple DDG in Figure 6. Assume that each of the three instructions use the same resource each with resource delays of two units, and that two resources were available. The lower bound IIII would be three (from $(\frac{2+2+2}{2})$). However, there is no possible way to place all three instructions into a resource table with three time slots and maintain the resource delay requirements (see Figure 7.a). As a result, the IIII must be increased above the calculated lower bound to four time units (see Figure 7.b).

Note that in the reservation tables of Figure 7, the time value is calculated with respect to the starting time of the loop modulo the IIII. The instruction schedule is then repeated every IIII time units.

In those cases where the resource delays are not of unit length, loop unrolling prior to Modulo Scheduling can result in reducing the final IIII to a value closer to the lower bound IIII. Enough unrolling will result in achieving the minimal IIII. For example, unrolling the loop having the code of Figure 6 one time will result in the DDG (actually a forest) of Figure 8.a. The calculated IIII is now six time units, which will satisfy the needs for the resource delay (see Figure 8.b).

The overall effect is that two of the original iterations can now be executed in an average time of three tine units each, which was the original lower bound on the IIII.

Because loop unrolling can be used to overcome the problem with resource delays, with no loss of generality, we will assume that the resource delay for all instructions is one time unit. In this manner, the schedule produced by the table is guaranteed to result in optimal utilization of those resources for the loop instructions, restricted only by limitations of the most used resource.

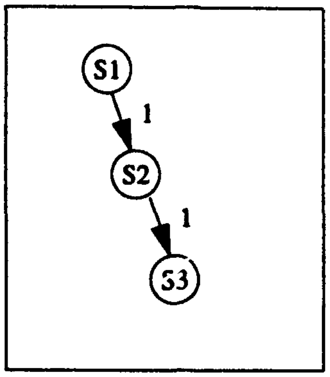


Figure 6: Simple Acyclic DDG for Loop Code with Three Instructions

time(mod 3)	resource 1	resource 2
0	S1	
1		S2
2	S3	

With the three instructions, at least two must be scheduled on the same resource. But with resource delay of two time units, an IIII of three time units is not adequate.

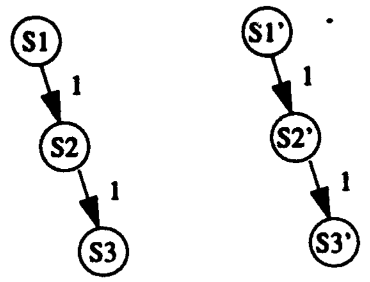
a. Reservation Table with Inadequate IIII of Three Time Units

time(mod 4)	resource 1	resource 2
0	S1	
1		S2
2	S3	
3		

By increasing the IIII to four time units, the instructions can be scheduled and meet the resource delay requirements.

b. Reservation Table with Adequate IIII of Four Time Units

Figure 7: IIII Adjustment to Meet Resource Delay Requirements



Two original iterations now contained in the inner loop. The acyclic nature of the original DDG results in two independent trees representing the new unrolled loop code. The second iteration instructions are identified with " ' " to denote that it is not the same iteration instruction

a. Data Dependence Forest for Unrolled Loop

time(mod 4)	resource 1	resource 2
0	S1	
1		S2
2	S3	
3		S1'
4	S2'	
5		S3'

b. Modulo Resource Reservation Table For Unrolled Loop

Figure 8: Unrolled Loop DDG and Reservation Table

The Modulo Resource Reservation Table of Figure 9 is generated for the DDG in Figure 4. The time value is given with respect to the starting time of the loop modulo four (since the IIII is four) and the schedule is therefore repeated every four time units.

	Resource Unit			
time(mod 4)	adder	adder	multiplier	Load/Store
0	S5	S7	S1	S9
1	S2	S11	S6	S12
2	S3	S10		
3	S8			S4

Figure 9: Modulo Resource Reservation Table for DDG of Figure 4

The reduced execution time from the original loop to the loop using loop pipelining is due to the overlapping of instructions from different iterations. The dependences which do exist between the instructions determine the relative iteration to which each of the instructions in this schedule belongs.

Once the table has been created, identifying the proper relative iteration for each instruction in the reservation table is the only sub-step in the Modulo Scheduling procedure which any complexity. Letting "k" indicate the iteration, then the appropriate dependences between instructions are satisfied if the iterations are labeled as in Figure 10. In this case, the time is given with respect to the starting time of the loop, being t_0. The value "a" represents the number of iteration executed prior to the pipelined loop, to meet the preconditioning or prolog requirements as explained in Chapter IV. The relationship between the instructions' subscripts are determined by the latency and loop delay requirements represented in the DDG.

The Modulo Resource Reservation Table, as shown in Figure 10, can be used to generate the restructured loop body for a VLIW machine with the given resources. The simplicity of this technique allows easy automation, with performance benefits scalable with added resources.

	Resource Unit			
time	adder	adder	multiplier	Load/Store
$4(k\text{-}a)+t_0$	$(S5)_k$	$(S7)_k$	$(S1)_k$	$(S9)_{k-1}$
$4(k\text{-}a)+t_0+1$	$(S2)_k$	$(S11)_k$	$(S6)_k$	$(S12)_{k-2}$
$4(k\text{-}a)+t_0+2$	$(S3)_k$	$(S10)_{k-1}$		
$4(k\text{-}a)+t_0+3$	$(S8)_k$			$(S4)_k$

Figure 10: Modulo Resource Reservation Table for DDG of Figure 4 with Relative Iterations Identified

B. MODULO SCHEDULING OF CYCLIC DDGs

One of the greatest limitations to the use Modulo Scheduling is the complexity and inefficiency when considering data dependence cycles. As with modulo scheduling of acyclic DDGs, the intent is to create a pipelined kernel schedule which has the smallest IIII possible. However, while the equation for the lower bound on the IIII presented in the last section only considers the restrictions which are placed on the IIII due to resource availability, the existence of dependence cycles creates additional constraints on the lower bound of IIII that must be met. For example, consider the simple cyclic DDG of Figure 11, which represents the loop code instructions of a loop with a cyclic dependence between the instructions S1 and S3.

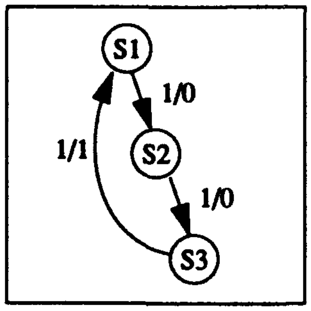


Figure 11: Simple Cyclic DDG

Assume that two fully capable functional units are available to execute the instructions, and, as said before, all resource delays are one time unit. The lower bound on the IIII due to the resources is two time units. However, the path of the cycle requires that it take at least three time units between execution S1 in one iteration and the execution of S1 in the next iteration (this is actually true for any instruction in the graph). The result is a lower bound on the IIII due to the cyclic dependence which is more restrictive than the lower bound due to resource availability. By requiring an IIII of three time units, the additional resource available does not aid in improving performance. In fact, no matter how many resources are made available (two, ten, one hundred, etc.), while the lower bound on the IIII due to the resources decreases to a potential value of one, the performance is limited by the unyielding bound placed on the IIII by the dependence cycle.

As an additional example, consider again the cyclic DDG shown in Figure 2, having a a cyclic dependence from node S12 to S4. Assuming that there are the four resources as in the previous section (two adders, a multiplier, and a load/store unit), then the lower bound of IIII based on resource constraints would again yield a value of four time units. With no cyclic dependences, a Modulo Resource Reservation Table could be generated with this IIII as in the previous section, with maximum resource utilization for the loop. However, the cyclic dependences require that the lower bound on IIII be five time units (as derived using techniques such as those described by Zaky [Ref. 7]). Again, the result is poorer performance and under-utilization of the resources.

In this case as always, because the lower bound on the IIII due to the cyclic dependences is independent of the available resources, no improvement on the performance can be obtained by adding resources. That is, as long as the cyclic dependences require a more limiting IIII, performance improvement is **not scalable** with added resource, as in the acyclic DDG case.

C. PIPELINING OF PERFECTLY-NESTED LOOPS

For the case of perfectly-nested loops, Modulo Scheduling methods can certainly be applied directly to the innermost loop of a nested loop structure. If no cyclic dependences exist between the iterations of the innermost loop of a perfectly-nested loop structure (i.e., there are no innermost loop carried dependences causing cycles), the innermost loop can be pipelined using the efficient acyclic DDG Modulo Scheduling method.

In cases where there are cyclic dependences across the innermost loop, loop interchange techniques can sometimes be used to restructure the loop to transfer the loop carried dependences of the innermost loop to the outermost loop. In some cases, this can create totally independent innermost loop iterations.

For example, consider the loop shown below:

```
for $i_1$ in 1..$N_1$ loop
    for $i_2$ in 1..$N_2$ loop
        $A(i_1,i_2)= 3*A(i_1, i_2-1)$
    end loop
end loop
```

For the same outer loop iteration, the statement in the innermost loop is dependent on the same statement from the previous innermost loop iteration, thus forming a cycle in the DDG.

By applying a transformation that interchanges the loops, the resultant code would be:

```
for $i_2$ in 1..$N_2$ loop
    for $i_1$ in 1..$N_1$ loop
        $A(i_1,i_2)= 3*A(i_1, i_2-1)$
    end loop
end loop
```

The interchange transfers the loop carried dependence to the outermost loop, leaving a parallel innermost loop to which the Acyclic DDG Modulo Scheduling method can be applied.

Unfortunately, many loops contain data dependence cycles which carry across all dimensions of the loop, for example:

```
for $i_1$ in 1..$N_1$ loop
    for $i_2$ in 1..$N_2$ loop
        $A(i_1,i_2)= A(i_1, i_2-1)+A(i_1-1, i_2)$
    end loop
end loop
```

In this case, a two cyclic dependences exists due to loop carried dependences across both the innermost and outermost loop boundaries. The interchange of the loop structures transfers the original innermost loop carried dependence to the outermost loop, and the original outermost loop carried dependence to the innermost loop. The same situation exists with a cyclic dependence in the innermost loop. As a result, the simple Acyclic DDG Modulo Scheduling method cannot be applied. Certainly, a cyclic DDG Modulo Scheduling method could be applied, before or after the interchange. However, it would be beneficial if the constraints imposed by cyclic dependences could be altogether avoided. Unfortunately, no alternative method for loop pipelining has yet been proposed which will eliminate the restrictions of cyclic dependences in this and similar cases.

A major motivation, therefore, for creating the loop pipelining technique presented in this thesis is to provide an alternative method to loop pipelining of perfectly-nested loops, which when faced with the problem above, will circumvent the problems of cyclic dependences and guarantee the applicability of the Modulo Scheduling for acyclic graphs to the innermost loop.

III. AN OVERVIEW OF THE PROPOSED LOOP PIPELINING TECHNIQUE

This chapter describes the general technique for loop pipelining of a perfectly-nested loop structure developed for this thesis The intent of the technique is to provide a means for loop pipelining the innermost loop of perfectly-nested loop structures which have cyclic dependences. Unlike previously presented loop pipelining techniques, however, this technique overcomes the performance restrictions which cyclic dependences can impose, while specifically targeting the resultant execution schedule for a particular set of resources.

The technique requires the use of two basic tools, both of which have previously been developed separately, but when combined, create a powerful technique for loop pipelining. ***It is the combination of the two tools which is unique to the loop pipelining technique presented in this thesis.***

The first tool is a linear transformation method which restructures any original perfectly-nested loop structure into one with a parallel innermost loop--that is, one with totally independent innermost loop iterations. With the removal of all cyclic dependencies, the resultant loop code DDG can then be loop pipelined with the application of the second tool, the Acyclic DDG Modulo Scheduling method previously discussed. The final result will be a pipelined kernel schedule with which a restructure innermost loop can be created for execution on the target VLIW type machine. Each of these tools are described in the sections below.

The loop pipelining technique described considers only perfectly nested loops with unit step increases in control variables. Loops with step increments greater that one can be normalized to create loops with unit step increases and with index lower bounds equal to one. While the technique is applicable to n-nested loops, the technique only *requires* the alteration of the structure of the two innermost loops.

In general, the loop structures to which this method is applied have the form:

```
for $i_1$ in 1..$N_1$ loop
    for $i_2$ in 1..$N_2$ loop
            .
            .
            .
            for $i_{n-1}$ in 1..$N_{n-1}$ loop
                for $i_n$ in 1..$N_n$ loop
                    (original loop body)
                end loop
            end loop
            .
            .
            .
    end loop
end loop
```

A. TRANSFORMATION OF THE ORIGINAL LOOP STRUCTURE

The first step in the loop pipelining technique proposed in this thesis is the application of a loop transformation on the original loop structure. In Section II.C, it was seen that for some perfectly nested loops structures, a loop interchange would be sufficient to eliminate innermost loop cyclic dependencies and allow the application of the acyclic DDG Modulo Scheduling Technique. The problem, as was noted, is the fact that loop structures exist which carry loop dependencies across multiple loop boundaries, creating dependence cycles which cannot be eliminated with mere loop interchanges. In fact, the scope of the problem is extended to those loops which cannot directly support an interchange in any case. For example, consider the two dimensional loop structure below:

```
for $i_1$ in 1..$N_1$ loop
    for $i_2$ in 1..$N_2$ loop
        $A(i_1,i_2)= A(i_1, i_2-1)+A(i_1-1, i_2+1)$
    end loop
end loop
```

This loop not only has cyclic dependencies across both loops, but interchanging the two loops structures would alter the semantics of the structure. Interchange, therefore, cannot be directly applied.

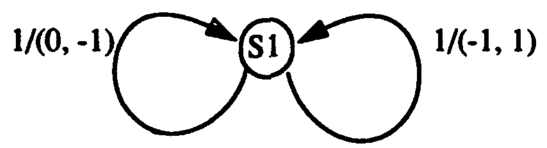
However, transformations do exist that first skew of the innermost loop, and then apply a loop interchange to once again produce parallel innermost loop iterations. The general method using this process to produce parallel innermost loop iterations is referred to as the *Wavefront Method* (or Hyperplane Method) and addressed by Lamport [Ref. 10],as well as by Wolf and Lam [Ref. 11].This method is described below, followed by the specific application to the loop pipelining method.

1. Explanation Of The Wavefront Method

The wavefront method of transformation was the ideal transformation method to use as the first step in the loop pipelining technique created. To understand the wavefront method, consider the two dimensional loop example shown in Figure 12.a. The DDG associated with this loop structure can be represented as in Figure 12.b.[1] For the purpose of this example, a latency of "one" is assigned to the addition instruction.

```
for $i_1$ in 1..100 loop
    for $i_2$ in 1..500 loop
        S1: $A(i_1,i_2)= A(i_1, i_2-1)+A(i_1-1, i_2+1)$
    end loop
end loop
```

a. Two Dimensional Loop



b. Associated DDG

Figure 12: Simple Two Dimensional Loop Structure With DDG

In this case, as in the case for all multi-dimensional loops, the loop delay identifier on the dependence arc is represented as a vector, with each element of the vector

1. For the purposes of this example, the loop body description will be left in high level representation as shown. In reality, the level at which the transformation and the modulo scheduling will be applied is a machine code level. At this point, a higher level representation of the loop structure and the DDG is used to simplify the explanation.

corresponding to one of the loop dimensions. In general, the vector is in the form (d_1, d_2, d_3, d_4,... d_n), where d_1 correspond to the delay associated with the outermost loop, and d_n corresponds to the delay associated with the innermost loop.

For the example, the two delay vectors (0, -1) and (-1, 1) refer to the dependences between the computation of an array value $A(i_1,i_2)$ and its use in the computation as the value $A(i_1, i_2-1)$ and $A(i_1-1, i_2+1)$, respectively.

The relationship between the iterations of the loop can be shown using a iteration space diagram, as in Figure 13. Each point in the space represents one loop code iteration, and the arcs between iteration points represent loop carried dependences between the iterations. Figure 13 represents the two dimensional iteration space diagram corresponding to the loop structure of Figure 12. The arcs continue uniformly throughout the iteration space in the same pattern as displayed.

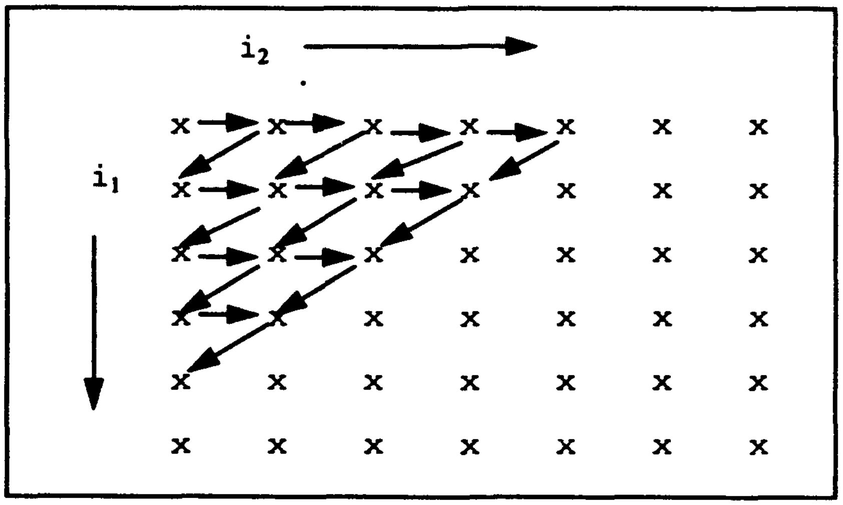


Figure 13: Iteration Space Diagram, Showing Iteration Dependences

For the example, loop carried dependences exist across both dimensions. In addition, the loops cannot be directly interchanged without changing the semantics of the loop.

In the case of the example, although cyclic dependences may exist along any dimension, there can be identified sets of iterations which lie along regular lines, or *Wavefronts*, through the iteration space, which do not have dependence relationships. In fact, Wolf and Lam [Ref. 11] claim that for any loop structure with a constant component loop delay vectors, a wavefront can always be found.

For the example iteration space of Figure 13, Figure 14 shows one choice of wavefront for which all iterations on any wavefront line are totally independent. In particular, along any line of the wavefront, the loop carried dependences which created the DDG cycles do not relate any two iterations of the wavefront. If the original loop structure can be transformed into one in which the innermost loop contains the iterations belonging to one line of an independent wavefront, as Lamport [Ref. 10] claims is possible, then the innermost loop iteration will be fully independent, and the acyclic DDG Modulo Scheduling method can be applied to the new innermost loop.

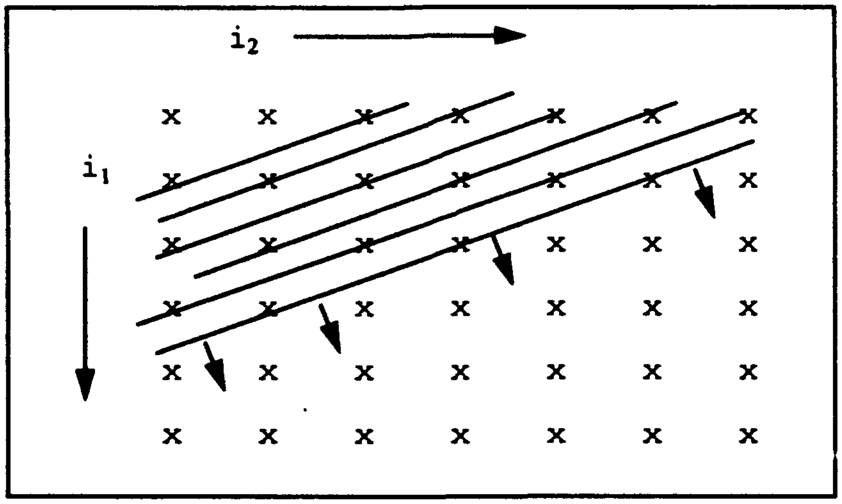


Figure 14: Iteration Space Diagram Showing a Wavefront for Independent Iterations

The necessary transformation accomplishing this task would have to skew the iteration space to "straighten out" the wavefront lines so that they fall along a single dimension, and then interchange the loop bounds to ensure the wavefront lines fall along the innermost dimension. The result of the skewing and loop interchange would produce a new iteration space with the shape of a parallelogram, as in Figure 15.

An in-depth discussion of the theory and application of applying the required loop transformation is presented by Wolf and Lam [Ref. 11]. The key is to perform a transformation which provides the desired affect while maintaining an execution order of the iterations which preserves the program intent. Wolf and Lam [Ref. 11] identify precisely the *unimodular transformation* (one that is performed by a square matrix with integer elements, and whose determinant is ±1) which produces the effect desired.

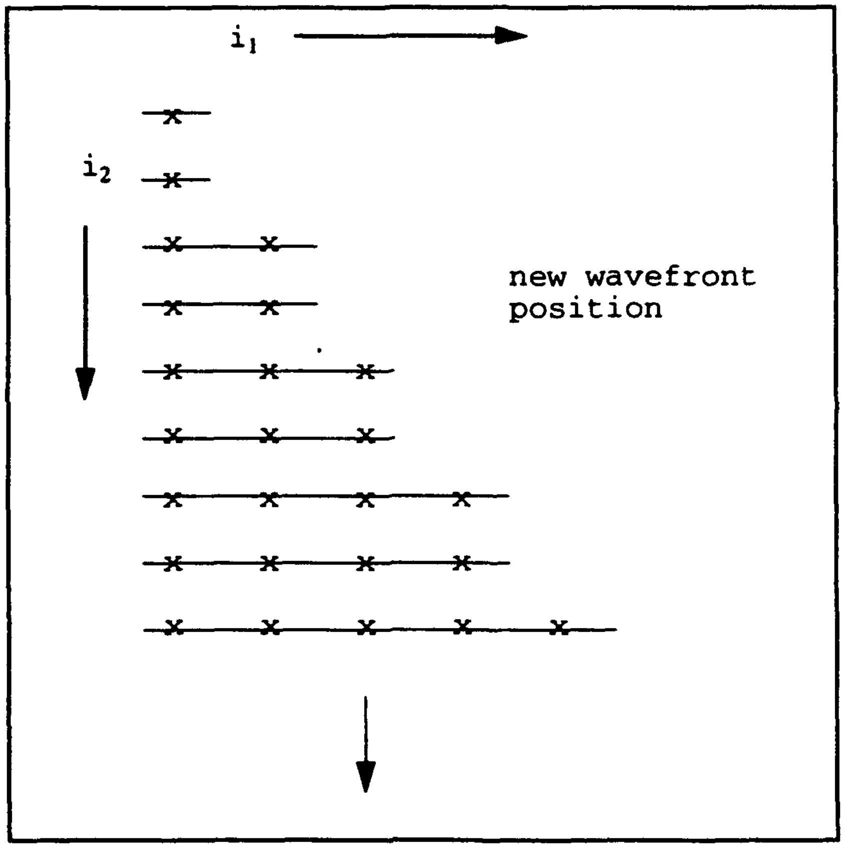


Figure 15: Transformed Iteration Space with Horizontal Wavefronts

When applied to the original loops structure, the unimodular transformation will produced a loop structure for which all loop delay vectors of the associated DDG have either the value of zero for the component of the vector associated with the innermost loop, or if this value is not zero, at least one other component value of the vector is non-zero. This will ensure that the innermost loop iterations for the transformed loop are independent, thus allowing the application of an Acyclic DDG Modulo Scheduling method to the innermost loop.

2. Determining The Transformation Matrix

Now that the basic motivation for and explanation of the wavefront transformation has been presented, the transformation process can be described. A transformation which guarantees that the restructured loop has a completely parallel innermost loop can be obtained in two steps: the first step is the skewing process and the second step is the interchange process. As mentioned earlier, the transformation method is discussed in detail by Wolf and Lam [Ref. 11], and is summarized below.

a. Step One: Obtaining The Skewing Matrix

The first step in the transformation is to apply skewing to the innermost loop, with respect to the second innermost loop, as necessary to ensure that the two innermost loops are ***fully permutable***--that is, to allow the innermost loop to be interchanged with the second innermost loop without altering the semantics of the loop.

For creating a permutable nest for the two innermost dimension, Wolf and Lam [Ref. 11] prescribe that the proper skewing is applied using a transformation matrix M_{skew} defined as in the following Equation 3.

$$M_{skew} = \begin{bmatrix} 1\,0\,0\,.\,.\,.\,0\,0 \\ 0\,1\,0\,.\,.\,.\,0\,0 \\ \cdot \\ \cdot \\ \cdot \\ 0\,0\,0\,.\,.\,.\,1\,0 \\ 0\,0\,0\,.\,.\,.\,sf\,1 \end{bmatrix} \qquad \text{(Eq. 3)}$$

The varaible *sf* is the called the **skewing factor**, with a value defined by the equation:

$$sf = \begin{cases} 0 & \text{if } \forall \vec{d}: d_n \geq 0 \\ \max_{(\vec{d} \mid (\vec{d} \in D \wedge d_n \neq 0))} \left\lceil \dfrac{-(d_n - 1)}{d_n} \right\rceil & \text{otherwise} \end{cases} \qquad \text{(Eq. 4)}$$

where D is the set of all loop delay vector in the original DDG

When M_{skew} is applied to the loop structure, it results in a skewed loop structure in which the two innermost loops are permutable. However, loop carried dependencies can still exist and cause the cyclic dependences which are not desired. The next step, therefore, is to create the parallel innermost loop.

For an example of this step, consider the DDG in Figure 12. For this example, the calculation of the *sf* (from equation 4) yields a value of 1. Hence, $M_{skew} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$.

b. Step Two: Creating The Parallel Innermost Loop

Determining the value of M_{skew} is only the first step in creating the transformation matrix. In order to guarantee that the innermost loop is parallel, the loop structure must be skewed one additional step beyond that skewing prescribed by M_{skew}. This will eliminate the existence of loop carried dependences which are solely across the second innermost loop. This skewing is combined with the interchange of the innermost loop with the second innermost loop. The result is a loop structure for which there is guaranteed no loop carried dependences which cross only the innermost loop. This, then, meets the requirements for having a fully parallel innermost loop.

Wolf and Lam [Ref. 11] describe the required additional transformation needed to make the (n-1) innermost nested loops of a n-dimensional loop structure parallel.

For the case of a two dimensional loop structure requiring the innermost loop to be fully parallel, the general case yields the transformation matrix $M_{\text{skew-interchange}}$, defined as:

$$M_{skew-interchange} = \begin{bmatrix} 1\,0\,0 \ldots 0\,0 \\ 0\,1\,0 \ldots 0\,0 \\ \cdot \\ \cdot \\ \cdot \\ 0\,0\,0 \ldots 1\,1 \\ 0\,0\,0 \ldots 1\,0 \end{bmatrix} \qquad \text{(Eq. 5)}$$

Once again using the DDG in Figure 12, as an example, $M_{skew-interchange} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$

c. Combining The Steps

Once the addition skewing and interchange matrix is obtained, the entire transformation process can be performed all in a single step using the product matrix M_{final}, calculated as:

$$M_{final} = M_{skew-interchange} \times M_{skew} = \begin{bmatrix} 1\,0\,0 \ldots & 0 & 0 \\ 0\,1\,0 \ldots & 0 & 0 \\ & \cdot & \\ & \cdot & \\ & \cdot & \\ 0\,0\,0 \ldots & 1+sf & 1 \\ 0\,0\,0 \ldots & 1 & 0 \end{bmatrix} \qquad \text{(Eq. 6)}$$

Important to note that the total skewing applied is given by the factor (sf + 1). Also important is the fact that the determination of M_{final} does not need the intermediate calculations of M_{skew} and $M_{\text{skew-interchange}}$, but can be determined immediately once the value of the sf is known.

Continuing with the previous example, the resultant final transformation matrix is $M_{final} = M_{skew-interchange} \times M_{skew} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}$

3. Transforming The Original Loop Structure

Once the final transformation matrix has been determined, it can be used to transform the index space from the original loop structure to the new loop structure with desired parallel inner loop iterations. Two direct results occur due to the transformation: first, the loop structure changes creating new loop index variables as functions of the original index variables; and second, the DDG is transformed into a DDG on which acyclic Modulo Scheduling can be applied to the innermost dimension.

a. Transforming The Loop Code

The first step in transforming the original loop into the final loop is to apply the transformation to the loop code. This transformation affects the loop code in two ways: first, it requires the addition of transformation instructions which act as "mending" code at the beginning of the new innermost loop to calculate the values of the variables which were original index variables, and second, it determines the change in loop boundaries for the new code.

(1) Adding The Transformation Instructions. The additional code which must be included in the body of the innermost loop is determined directly from the inverse of the final transformation matrix. The transformation from the old index space the to the new index space uses M_{final}, and is represented by Equation 7.

$$\begin{bmatrix} i'_1 \\ i'_2 \\ \cdot \\ \cdot \\ \cdot \\ i'_{n-1} \\ i'_n \end{bmatrix} = \begin{bmatrix} 1\,0\,0\ldots & 0 & 0 \\ 0\,1\,0\ldots & 0 & 0 \\ & \cdot & \\ & \cdot & \\ & \cdot & \\ 0\,0\,0\ldots & 1+sf & 1 \\ 0\,0\,0\ldots & 1 & 0 \end{bmatrix} \times \begin{bmatrix} i_1 \\ i_2 \\ \cdot \\ \cdot \\ \cdot \\ i_{n-1} \\ i_n \end{bmatrix} \qquad \text{(Eq. 7)}$$

The mending code which is required for calculating the values of the variables which were original index variables can be found using the inverse transformation matrix, and is given by the equation:

$$\begin{bmatrix} i_1 \\ i_2 \\ \cdot \\ \cdot \\ \cdot \\ i_{n-1} \\ i_n \end{bmatrix} = \begin{bmatrix} 1\,0\,0\ldots 0 & 0 \\ 0\,1\,0\ldots 0 & 0 \\ \cdot & \\ \cdot & \\ \cdot & \\ 0\,0\,0\ldots 0 & 1 \\ 0\,0\,0\ldots 1 & (-(1+sf)) \end{bmatrix} \times \begin{bmatrix} i'_1 \\ i'_2 \\ \cdot \\ \cdot \\ \cdot \\ i'_{n-1} \\ i'_n \end{bmatrix} \qquad \text{(Eq. 8)}$$

As noted before, and as is made obvious here, only the two innermost dimensions are affected by the transformation. As a result, the above equations indicate that the only required additional code to be included in the innermost to complete the transformation are given by the equations: $i_{n-1} = i'_n$ and $i_n = i'_{n-1} - (1 + sf)\, i'_n$

The equality of i_{n-1} and i'_n helps simplify the situation by allowing the variable i'_n to be directly substituted for the i_{n-1} variable in the instructions. The only calculation required due to the transformation is for the variable i_n. This reduces the additional instructions required to only the second equation above, which is a calculation which then must be done at runtime.

For the example from Figure 12, the resultant transformation equation is therefore $i_2 = i'_1 - 2i'_2$.

The necessary addition of this equation to the innermost loop code is not specifically mentioned by Wolf and Lam [Ref. 11]. Although the relationship between the variables is clearly identified, the particular implementation and necessary overhead required by the transformation is not addressed. Because we are also concerned with the practical aspects involved in the generation of a code following the application of the loop pipelining technique, the inclusion of the transformation instructions in the loop body cannot be overlooked and is vital to the proper implementation of the technique. In addition, the added code implies the addition of overhead to the technique which must be considered when evaluating the effectiveness of the technique.

(2) Changing The Loop Bounds. In addition to adding transformation instructions into the loop body, the loop boundaries for the loop control variables must also be altered to conform to the new loop variables. Wolf and Lam [Ref. 11] specifically address the effect of the transformation on the loop bounds, which now are dependent upon the skewing applied, the loop interchange, and the original loop bounds.

Again, because the transformation only affects the two innermost loops, the general n-dimensional discussion provided by Wolf and Lam [Ref. 11] is simplified for the two dimensional case of interest. Only the bounds on the two innermost loop variables require adjustment. The bounds on all other loop variables remain the same.

The bounds on the new two innermost loop variables are calculated based on the value of *sf* and the original loop boundaries. In general, the range of the second innermost loop variable, i'_{n-1}, becomes $(sf+2) \ldots [(sf+1) \times N_{n-1} + N_n]$.

The range of the innermost loop variable, i'_n, now becomes $max(1, \left\lceil \frac{i'_{n-1} - N_n}{sf+1} \right\rceil) \ldots min(\left\lfloor \frac{i'_{n-1} - 1}{sf+1} \right\rfloor, N_{n-1})$.

Both boundaries require some calculation. The boundaries for the second innermost loop are based only on the *sf* and the original loop bounds. Because these are known during the transformation process (compile time), these boundaries can be calculated prior to run time. The innermost loop boundaries, however, also depends on the value of i'_{n-1}, and must be calculated at run time. Fortunately, this calculation can be done outside of the innermost loop (but within the second innermost loop) not adding additional code to the innermost loop.

Once again using the example from Figure 12, the bounds on second innermost loop variable, i'_1, becomes 3..700. and the range of the innermost loop variable, i'_2, now becomes $max(1, \left\lceil \frac{i'_1 - 500}{2} \right\rceil) \ldots min(\left\lfloor \frac{i'_1 - 1}{2} \right\rfloor, 100)$.

(3) The Final Transformed Loop Code. The loop code transformation is complete with the combination of the addition of the transformation equations and the loop control variable boundary calculations. For the general case, the resultant transformed structure from the final two steps is

```
for $i_1$ in 1..$N_1$ loop
    for $i_2$ in 1..$N_2$ loop
              .
              .
              .
          for $i'_{n-1}$ in $(sf+2) \ldots [(sf+1) \times N_{n-1} + N_n]$ loop
              for $i'_n$ in $max(1, \left\lceil \frac{i'_{n-1} - N_n}{sf+1} \right\rceil) \ldots min(\left\lfloor \frac{i'_{n-1} - 1}{sf+1} \right\rfloor, N_{n-1})$
                  loop
                  $i_n := i'_{n-1} - (1+sf) i'_n$
                  (original loop body)
              end loop
          end loop
              .
              .
              .
    end loop
end loop
```

Once more, for the example in Figure 12, the variable i'_2 is directly substituted for the i_1 variable in the loop body statement. In addition, the transformation equation for calculating the i_2 variable is added to the inner loop and the loop control variables and boundaries are altered to maintain the semantics of the loop. As a result, the final transformed loop structure would have the form:

```
for i'_1 in 3..700 loop
    for i'_2 in $max(1, \left\lceil \frac{i'_1 - 500}{2} \right\rceil) ... min(\left\lfloor \frac{i'_1 - 1}{2} \right\rfloor, 100)$ loop
        S2: $i_2 = i'_1 - 2i'_2$
        S1: $A(i'_2, i_2) = A(i'_2, i_2 - 1) + A(i'_2 - 1, i_2 + 1)$
    end loop
end loop
```

b. Transforming The Loop DDG

Although the transformation of the loop code is important in determining the transformed loop body and the new loop control variable boundaries, the wavefront transformation must be applied to the original loop DDG to obtain the dependence graph that represents the innermost loop which will be modulo scheduled. This involves applying the transformation to the loop delay vectors of the original DDG by using the transformation matrix and adding the two additional instructions used to calculate the value of i_2 to the DDG.

(1) Altering The Delay Vectors. The delay vectors associated with each arc of the DDG are altered using the transformation matrix M_{final}. Again, only the components of the delay vector corresponding to the two innermost loop dimensions are altered. The new delay vectors for the transformed DDG can be labelled as $\vec{d}'$.

(2) Adding The Transformation Instructions. The transformation instructions which were discussed in Section III.A.3.a.1 are added to the DDG by determining the dependences which exist between the new instructions and those original instructions which use the original i_n variable value. Nodes are then added to represent the

transformation instructions, and appropriate arcs are attached to incorporate the needed dependences.

(3) The Resultant DDG. Once transformed in the above manner, the DDG which represents the innermost loop alone is obtained by eliminating all arcs from the Transformed DDG for which

$$\vec{d}'(x) \neq 0, \text{ where } x \text{ is any value } 1..n\text{-}1 \qquad \text{(Eq. 9)}$$

Because the transformation was designed to make all innermost loop iterations parallel, the result will be the elimination of all arcs which represent any loop carried dependences. The resultant DDG will be referred to as the *Modified Transformed DDG*.

(4) Example. For an example of the process described above, again consider the DDG of Figure 12. The two loop delay vectors are modified by multiplying these vectors by the M_{final} matrix (recall that $M_{final} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}$). The transformation equation is then added to the DDG (the transformation equation was labelled as S2 in the example code). The last step in the DDG transformation was to eliminate all arcs which represent loop carried dependences across the outer loops. The resultant DDG transformation follows the steps shown in Figure 16. The latency associated with the dependence between the S2 instruction and the S1 instruction is arbitrarily set to one in this case, again for ease of illustration.

4. Applying The Wavefront Method To Machine Code Loop Bodies

As previously noted, the example used to explain the wavefront method was simplified for ease of explanation. In reality, the intent of loop pipelining is to combine machine instructions into a single VLIW machine instruction. As a result, application of the wavefront method should be approached assuming that the machine instructions are

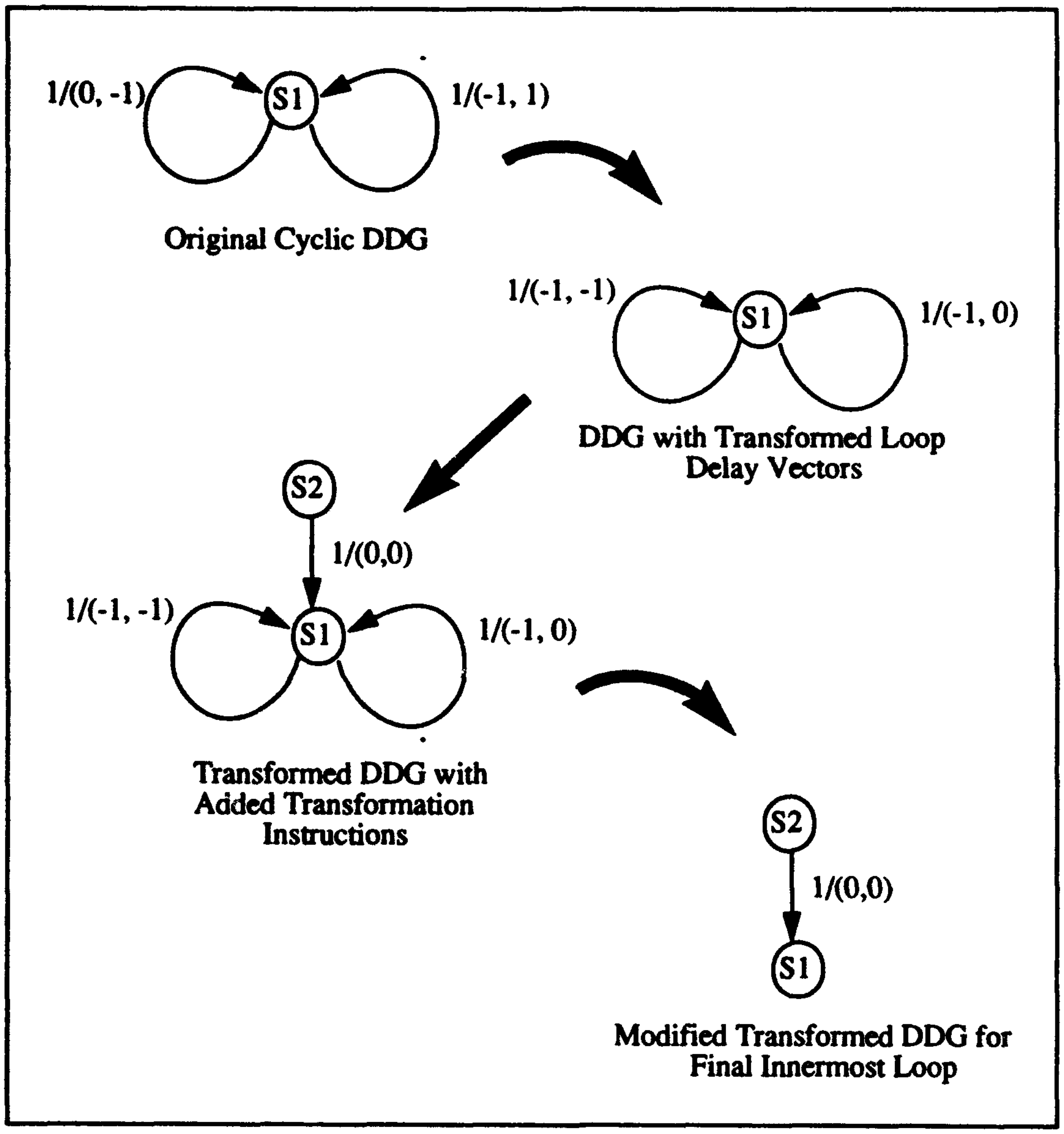

1/(0, -1)
S1
1/(-1, 1)
Original Cyclic DDG
1/(-1, -1)
S1
1/(-1, 0)
DDG with Transformed Loop Delay Vectors
S2
1/(0,0)
1/(-1, -1)
S1
1/(-1, 0)
Transformed DDG with Added Transformation Instructions
S2
1/(0,0)
S1
Modified Transformed DDG for Final Innermost Loop

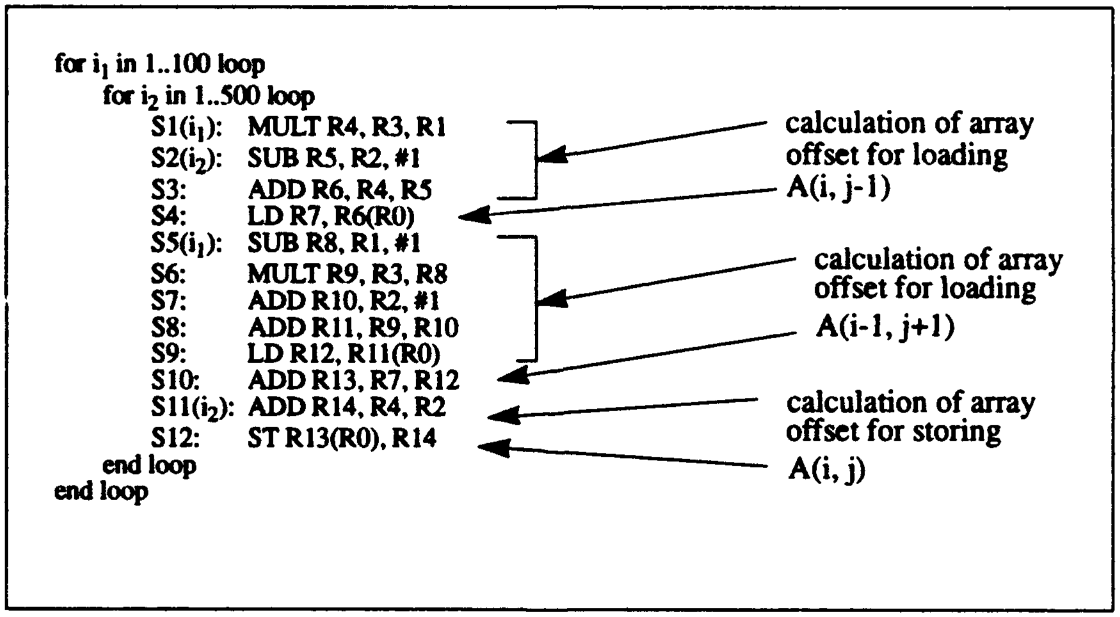
Figure 16: Modification Process of Original DDG

identified, and the nodes of the DDG which represents the loop body identify the individual machine instructions of the loop body.

For example, assume that the machine of concern is a RISC type, load/store machine. In addition, array variables are accessed in row major manor. Then, the loop example originally presented as

```
for $i_1$ in 1..100 loop
    for $i_2$ in 1..500 loop
        $A(i_1,i_2)= A(i_1, i_2-1)+A(i_1-1, i_2+1)$
    end loop
end loop
```

now becomes the loop structure of the form shown in Figure 17. In this case, statements which rely on the value of the loop control variable are indicated as doing so..



```
for $i_1$ in 1..100 loop
    for $i_2$ in 1..500 loop
        S1($i_1$):  MULT R4, R3, R1
        S2($i_2$):  SUB R5, R2, #1
        S3:         ADD R6, R4, R5
        S4:         LD R7, R6(R0)
        S5($i_1$):  SUB R8, R1, #1
        S6:         MULT R9, R3, R8
        S7:         ADD R10, R2, #1
        S8:         ADD R11, R9, R10
        S9:         LD R12, R11(R0)
        S10:        ADD R13, R7, R12
        S11($i_2$): ADD R14, R4, R2
        S12:        ST R13(R0), R14
    end loop
end loop
```

calculation of array offset for loading
A(i, j-1)
calculation of array offset for loading
A(i-1, j+1)
calculation of array offset for storing
A(i, j)

Figure 17: Extended Code For Figure 12 Example

In this example, the registers R0 is used as the base register for the array A(i,j). The register R1 is used to store the value of the i_1 variable, while the register R2 is used to store the value for the i_2 variable. The register R3 is used to store the length of each row, and would have the value of 500. Other registers are assigned as necessary to complete the calculation.

With these instructions, the corresponding DDG is shown in Figure 18. It is important to note that anti-dependency arcs which are due purely to the naming of registers are not included in the DDG. The treatment of such dependences is discussed at length in Section IV.A.

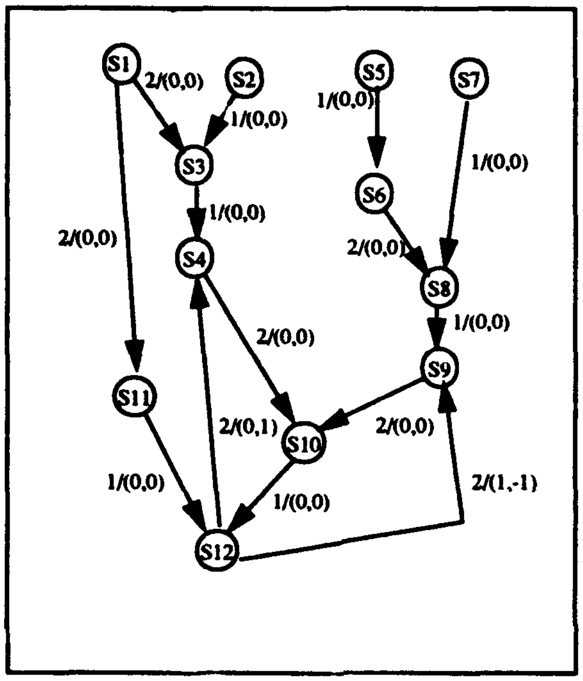


Figure 18: Cyclic DDG With Nodes Representing Machine Code Instructions

In this example, the node labelled S1 in Figure 12 is now expanded to identify the individual RISC type machine instructions which must be utilized to execute the loop body. The loop carried dependences which were displayed in Figure 12 are still represented by the dependence arcs between S12 and S4, and S12 and S9 in Figure 18. These dependences again create a cycles in the DDG.

The following sections indicate the result of applying the wavefront transformation procedure to the extended loop body with machine instructions.

a. The Transformation Matrix

The transformation matrix obtained for the extended example is precisely that obtained previously for the simplified example. That is, the *sf* is one, and $M_{final} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}$

b. Transformation Of The Loop

The transformation of the loop structures for the machine code example mirrors that for the simplified example.

(1) Transforming The Loop Structure. The transformation instructions which calculate the value of i_2 is still given by the formula $i_2 = i'_1 - 2i'_2$. However, this equation must be expressed in terms of machine instructions, and can be done so with two elementary calculations--one multiplication and one subtraction.

The loop bounds calculation is also independent of the exchange of the machine code instructions with the simplified loop body statement. As a result, if the instructions for the transformations are labelled S13 and S14, the transformed loop becomes:

```
for i'1 in 3..700 loop
    for i'2 in $max(1, \lceil \frac{i'_1 - 500}{2} \rceil) ... min(\lfloor \frac{i'_1 - 1}{2} \rfloor, 100)$ loop
        S13(i'2) MULT R16, #2, R1
        S14(i'1):SUB R2, R15, R16
        S1(i'2):  MULT R4, R3, R1
            .
            .
            .
```

As previously described, i'_2 replaces the variable i_1 in the loop code so that the value of i'_2 is now stored in register R1, as was i_1 originally. The value of i_2, once calculated, is placed in the R2 register as before. Additional registers R16 and R15 are needed for temporary storage in the calculation of i_2, and for storing the value of i'_1, respectively.

c. *Transforming The DDG*

Transformation of the DDG again follows the previously outlined procedure. The loop delay vectors of Figure 18 are altered using the transformed matrix, and instructions S13 and S14 are added to the DDG. The DDG representing only the innermost loop is isolated by removing from the transformed DDG all arcs which represent carry loop dependences across the outermost loop.

The the alterations to the DDG as described in Section III.A.3.b are displayed in Figure 19, which shows the original DDG, the transformed DDG with transformation instructions S13 and S14 and their dependence connections, and the final modified DDG for the final innermost loop.

B. APPLYING THE ACYCLIC DDG MODULO SCHEDULING METHOD

Application of the wavefront transformation to the original loop structure marks the end of the first major step required by the loop pipelining technique presented in this paper. At the conclusion of this step, the innermost loop of the transformed loop structure is free from dependence cycles.That is, the modified transformed DDG (the loop structure DDG representing only the innermost loop) is cycle free. This is exactly the conditions required to carry out the next major step of the presented loop pipelining procedure: the application of the acyclic DDG modulo scheduling method.

The procedure for creating the pipelined schedule for the transformed innermost loop follows precisely the Modulo Scheduling Method described Section II.A, originally presented by Rau and Glaeser [Ref. 4], with minor modifications. The Acyclic DDG Modulo Scheduling method can be used following the wavefront transformation because the transformation has produced independent innermost loop iterations. The resultant innermost loop DDG is, therefore, acyclic.

Various approaches and heuristics can be applied to properly fill the Modulo Resource Reservation Table using an Acyclic DDG Modulo Scheduling Technique. Work oy Hsu [Ref. 12] can be referred to for discussions of the various algorithms based on the desired

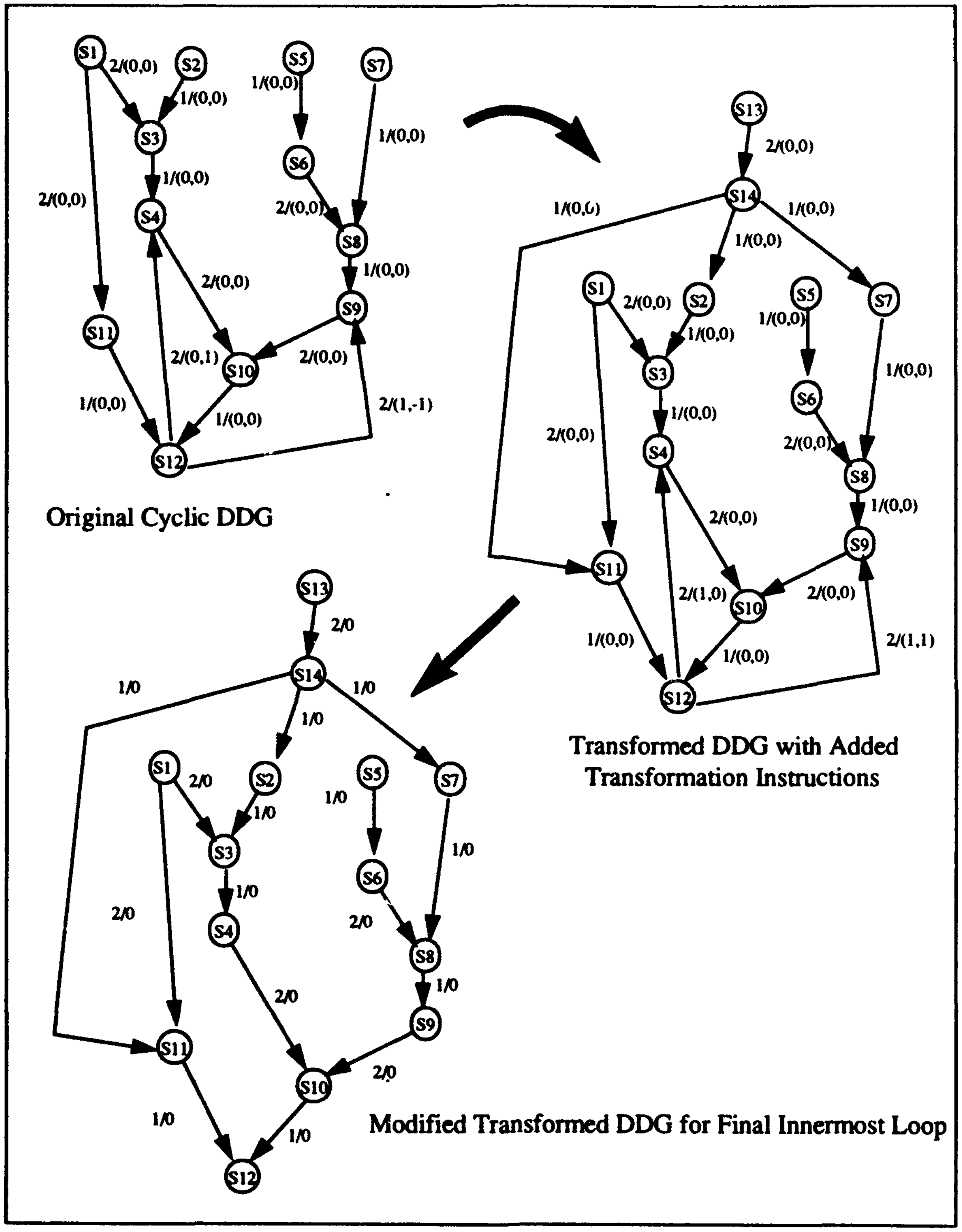

Original Cyclic DDG
Transformed DDG with Added Transformation Instructions
Modified Transformed DDG for Final Innermost Loop
S1
S2
S3
S4
S5
S6
S7
S8
S9
S10
S11
S12
S13
S14
2/(0,0)
1/(0,0)
2/(0,1)
2/(1,-1)
2/(1,0)
2/(1,1)
2/0
1/0

Figure 19: Modification Process of DDG with Machine Code Loop Body

result. In reality, the simplicity of the Acyclic DDG Modulo Scheduling Method actually allows a random placement of the instructions into the Modulo Resource Reservation Table, in the correct resource slots, of course. However, it might be desirable to create certain characteristics in the pipelined kernel, for example, minimizing the lifetime of any register used in the pipelined code.

One scheduling algorithm which is based on the algorithm discussed by Rau and Glaeser [Ref. 4], but which attempts to minimize the register lifetime of all registers, is shown in Figure 20..

```
Calculate the IIII
        --fill the reservation table per following
        perform a topological sort of the DDG, with priority given to the
                        nodes with the greatest height weighted by arc latency
        while there are nodes not scheduled loop
            pick the highest priority node per topological sort
            if node has no parent then
                $node.starttime = 0$
            else
                $node.starttime =$
                        $max_{parents}[parent.starttime + (arc\ latency)_{parent \rightarrow node} -$
                                $(arc\ loop\ delay)_{parent \rightarrow node} \times IIII]$
            if node is not a branch then
                until node is scheduled loop
                    $node.starttime = (node.starttime) mod (IIII)$
                    if proper resource is available for node at node.slottime in table, then
                        node is scheduled by reserving resource for node at slottime
                        $node.subscript = [k - (node.starttime) div (IIII)]$
                    else
                        $node.starttime = node.starttime + 1$
                    end if
                end loop
            else (node is branch)
                until node is scheduled loop
                    $node.starttime = (node.starttime) mod (IIII)$
                    if proper resource is available for node at node.slottime in table
                            and $(node.starttime) mod (IIII) = (IIII - 1)$, then
                        node is scheduled by reserving resource for node at slottime
                        $node.subscript = [k - (node.starttime) div (IIII)]$
                    else
                        $node.starttime = node.starttime + 1$
                    end if
                end loop
            end if
        end loop
```

Figure 20: Modulo Resource Reservation Scheduling Algorithm Which Attempts To Reduce Register Variable Lifetimes

As explained by Rau and Glaeser [Ref. 4], this procedure ensures that no more than one instruction is assigned to any one resource during any time slot. In addition, dependence relationships are maintained because relative starting time calculations are based on starting times of predecessor instructions, and consider required delays due to latencies and loop carried dependences. The calculation of the relative iteration to which an instruction belongs merely converts the timing relationship of instruction starting times to relative iterations

The reference made in the algorithm to a branch instruction is required in the case where the loop control instructions for the innermost loop are included in the pipelined schedule. This will be discussed in the next chapter.

Figure 21 shows the result of applying the modulo scheduling algorithm to the final modified transformed DDG of Figure 19. In this case, the IIII is again calculated to be four time units. The additional addition and multiplication instructions added to the loop body have, in general, the potential to effect the calculation of the IIII, but do not alter the result in this case. The time of instruction execution is again represented as it was in Figure 10.

	Resource Unit			
time	adder	adder	multiplier	Load/Store
$4(k-a)+t_0$	$(S5)_k$	$(S3)_{k-1}$	$(S13)_k$	
$4(k-a)+t_0+1$	$(S8)_{k-1}$	$(S10)_{k-2}$	$(S1)_k$	$(S4)_{k-1}$
$4(k-a)+t_0+2$	$(S14)_k$	$(S11)_{k-1}$	$(S6)_k$	$(S9)_{k-1}$
$4(k-a)+t_0+3$	$(S2)_k$	$(S7)_k$		$(S12)_{k-2}$

Figure 21: Modulo Resource Reservation Table

C. A REVIEW OF THE PROPOSED LOOP PIPELINING TECHNIQUE

The two tools presented, the wavefront transformation and the acyclic DDG modulo scheduling procedure, combine to form the basis of the technique developed in this thesis.

This combination has not been described in previous works, although it is simple and efficiently scheduled instructions onto resources with a minimum IIII.

The creation of the Modulo Resource Reservation Table only requires the use of the acyclic modified transformed DDG. Although the creation of this modified transformed DDG is made possible by the wavefront transformation, the general results of this transformation are, in fact, completely known prior to the actual performance of the transformation. For example, it is known that the transformation is designed to remove all innermost loop carried dependencies. Knowing this, all arcs in the original DDG which do not have a zero loop delay vector can be removed from consideration in the modified transformed DDG. Additionally, the transformation also adds one addition and one multiplication instruction to the innermost loop code body, to calculate the value of the i_n variable. The specific instructions must use the value of *sf* calculated in the transformation phase, however, the nodes can still be added to the modified transformed DDG without knowing this value. Although not yet discussed, the nodes required for the loop control can also be added to create a slightly different modified transformed DDG. This alteration in the procedure was ignored during this chapter to simplify the discussion, but will be presented in the next chapter. In addition, the transformation logically replaces of the i_{n-1} variable with the new index variable i'_n. In reality, this merely requires the use of the register originally assigned to i_{n-1} to now be used for the value of i'_n.

As a result, the actual creation of the Modulo Resource Reservation Table can be performed in parallel, if desired, with the calculations required for the transformation (*sf* and loop bounds). A flowchart which illustrates the overall procedure is shown in Figure 22.

The creation of the Modulo Resource Reservation Table, however, only marks the end of the first two major steps in the useful application of the technique developed. As Figure 22 shows, the third and final step is the generation of code from the results of the first two steps. Discussion of code generation is minimal or non-existent in all references reviewed concerning previous loop pipelining technique presentations. The required alteration of the code to a final product is, however, of practical importance and will be discussed in the next chapter.

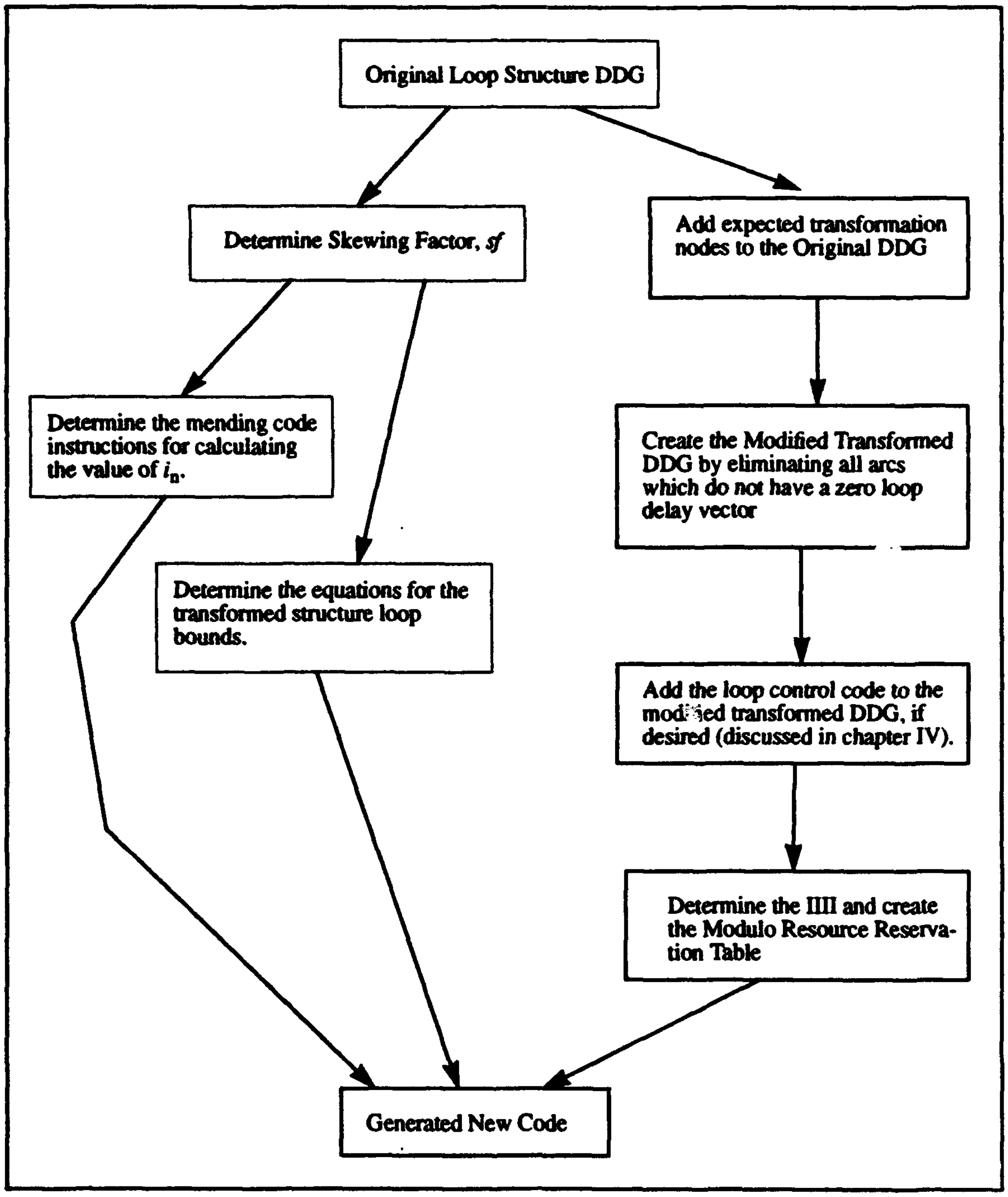

Original Loop Structure DDG
Determine Skewing Factor, *sf*
Add expected transformation nodes to the Original DDG
Determine the mending code instructions for calculating the value of i_n.
Create the Modified Transformed DDG by eliminating all arcs which do not have a zero loop delay vector
Determine the equations for the transformed structure loop bounds.
Add the loop control code to the modified transformed DDG, if desired (discussed in chapter IV).
Determine the IIII and create the Modulo Resource Reservation Table
Generated New Code

Figure 22: Proposed Loop Pipelining Technique Procedure Flowchart

IV. CODE GENERATION

Having performed the loop transformation and modulo scheduling, the final step of the presented loop pipelining technique is the code generation. Code generation depends, naturally, on the hardware support provided by the target machine. The hardware support that can aid in better performance goes beyond merely the number of resources. This section will first address the possible hardware capabilities of the target machine that can be used for supporting the modulo scheduling technique. The special considerations which must be addressed when generating the code are then reviewed. Lastly, the code generation procedure is described.

A. THE TARGET MACHINE HARDWARE

The procedure that was developed obviously was targeted for a VLIW type machine. The type of functional units that are provided by the machine can vary, and no abnormal limitations are placed on their capabilities. The type of units available determine, as was seen in Section III.B, the outcome of the Modulo Resource Reservation Table. The basic intent of the research done in this thesis, however, is to improve the overall performance capability by using VLIW machines. In that respect, additional machine hardware support designed specifically to support the modulo scheduling technique can only aid in realizing the fullest potential of the technique. Below is a description of the necessary and desired target machine hardware support that will be assumed when creating the final loop structure.

1. Basic Target Machine Requirements

The following assumptions are made concerning the target machines hardware support:

- The target machine processor is assumed to be RISC type processor, with multiple functional units capable of simultaneous execution of multiple instructions. The VLIW machine instruction word is comprised of a set of several instructions to be executed simultaneously, combined to form the VLIW instruction word. Each of the individual instructions making up a VLIW instruction will be referred to as VLIW *sub-instructions*, and can be represented by the instruction set similar to that of the DLX machine described by Hennessy and Patterson [Ref. 13]. The difference, of course, is that multiple-independent sub-instructions can be executed concurrently as part of a very long instruction word.

- A large number of registers are available for data storage, allowing the issue of register allocation to be ignored and addressed as a separate issue
- The memory sub-system for VLIW machines is a subject in itself. For the purpose of supporting the technique presented, it is assumed that an upper level memory sub-system exists, such as a cache, to support a single cycle access time assumed for load/stores. The issue of cache misses and hits will be addressed in a later chapter. Multiple Port cache memory is made available to allow concurrent Load/Store sub-instructions, accessing different memory locations, to be executed The procedure for scheduling instructions to avoid data dependency problems will preclude any instructions attempting to access the same memory location.

2. Additional Special Hardware Support

Additional special hardware support can be made available to better support the code generation concerns of modulo scheduling. Many of these hardware mechanisms are described by Rau, Schlansker, and Tirumalai [Ref. 6] as they pertain for use in modulo scheduling techniques. Although multiple supporting hardware components are described, the only two that will be assumed is the *Rotating Register File* (RRF) using the *Iteration Control Pointer* (ICP), and the *Iteration Control Register* (ICR) with support from the *Loop Counter* (LC) and *Epilog Stage Counter* (ESC).

A RRF is a file of multiple registers that can be accessed by a pointer reference to a single register in the rotating register file. The pointer can be the number identifier of the register desired. As a result, if a register file A[X] exist with 3 registers, then the registers can be referenced by referring to A[0], A[1], and A[2] (see Figure 23).

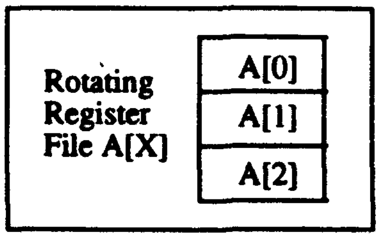


Figure 23: Simple Three Register Rotating Register File

Referencing can be made relative and variable with respect to some value y by referring to a register by A[y+constant], for example. The value of (y+constant) is evaluated in modulo the number of registers in the file to reference the correct register. For example, using the register file A[X] shown in Figure 23, the register file might be referenced in a loop as in the following:

```
for i in 1..10 loop
    use A[i]
end loop
```

In this loop, the registers in the register file A[X] will be referenced in a rotating manner, starting with A[1], and then A[2], A[0], A[1], A[2],...A[1].

The reference may also be some other mathematical expression, such as in the following:

```
for i in 1..10 loop
    use A[i+4]
end loop
```

In this case, the registers will be accessed in a rotating manner, starting with A[2] and ending with A[2].

To support the use of the rotating register file in the context desired, an ICP is used to identify the current iteration of some loop. It is originally set to zero, and a special loop control instruction will increment the ICP at the end of every iteration. The ICP can then be used as the variable to reference a register in an register file in some instruction. The special loop control instruction used to trigger the events will be called "brtop", which has as its argument the label for the top of the loop. The full use of the "brtop" instruction will be explained in a moment, but with respect to the ICP, the "brtop" instruction increments the ICP and causes a jump back to the top of the loop. For example, if the start of the loop is labeled "LOOP_TOP", then the above loop can be represented as follows:

```
LOOP_TOP:
    use A[ICP+4]
    brtop LOOP_TOP
```

Automatic incrementation of the ICP then allows the same instruction to reference the next register in the register file in the next iteration. This hardware support will become beneficial when dealing with the problem of register usage overlap discussed following sections.

The ICR is also a rotating register file with the specific purpose of providing for predicate execution of instructions. The ICR stores boolean values (actually one or zero) which can be referred to when evaluating the predicate for some instruction in the form "inst if p" (see Figure 24).

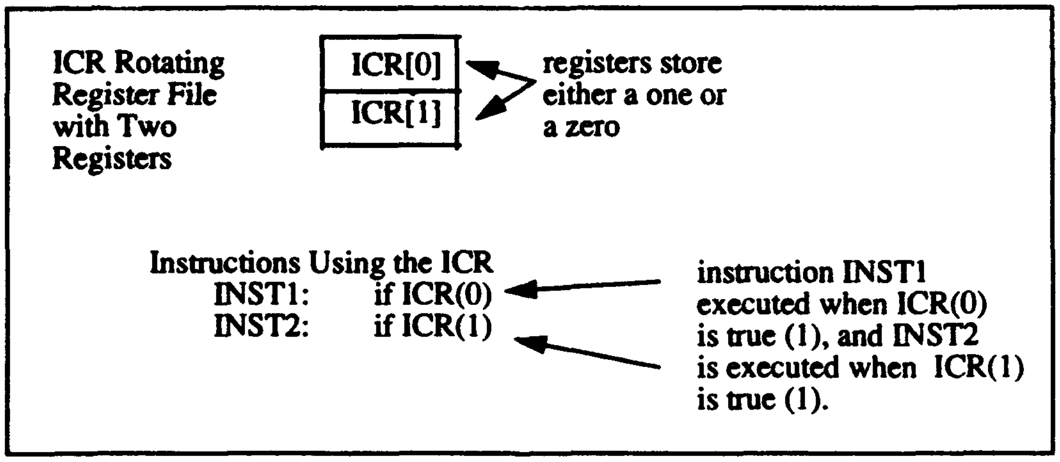

ICR Rotating Register File with Two Registers
ICR[0]
ICR[1]
registers store either a one or a zero
Instructions Using the ICR
INST1: if ICR(0)
INST2: if ICR(1)
instruction INST1 executed when ICR(0) is true (1), and INST2 is executed when ICR(1) is true (1).

Figure 24: Simple ICR Rotating Register File

The pointer for the current ICR register is originally set to zero and is incremented by one at the end of each loop iteration. This incrementation is triggered by the execution of the "brtop" instruction, just as is the incrementation of the ICP.

The current ICR register then changes at the end of each iterations. When selected as the current register, the value is set to either one or zero depending on the value of the LC. The LC keeps track of how many iterations are left to be started in the loop. It is originally set to the number of iterations desired, and is decremented at the end of each iteration, again by the execution of the "brtop" instruction. In this way, the LC is the hardware replacement for explicit loop control instructions.

The LC and the ECS counter work with the ICR to maintain special control of instruction execution. The ECS counter is initially set to one less than the number of registers in the ICR. As noted earlier, the LC is decremented with the execution of the "brtop" instruction. This is done prior to the incrementation of the ICR current register pointer. When LC is greater than zero, the ICR predicate register that becomes current is set to true (one). When the LC is zero, then the ICR pointer is reset to zero and the ESC counter activates. Also if the LC is less than or equal to zero, the ICR predicate register that becomes current is set to false (zero). Initially, the ICR pointer is set to zero and the value in the ICR(0) register is one. This will allow the execution of partial schedules of the modulo resource reservation table, which are needed in transitioning into the code as describe in the following sections. To aid in understanding the process described above, Figure 25 provides a flow chart depicting the major occurrences.

As previously mentioned, the special loop control instruction used to trigger the events will be called "brtop LOOP_TOP". The instruction first decrements ESC only if LC is less than or equal to zero, decrements the LC, and increments the ICP. The instruction then determines the action to be taken for the next ICR register. The control then jumps back to the top of the pipelined loop, labelled with the LOOP_TOP, unless both the LC and the ESC are less than or equal to zero. In this way, the branch is taken unless repetitions of the code executed was equal to the (original LC + original ECS). Figure 26 illustrates a simple example of the sequence of events for the used of each of these components.

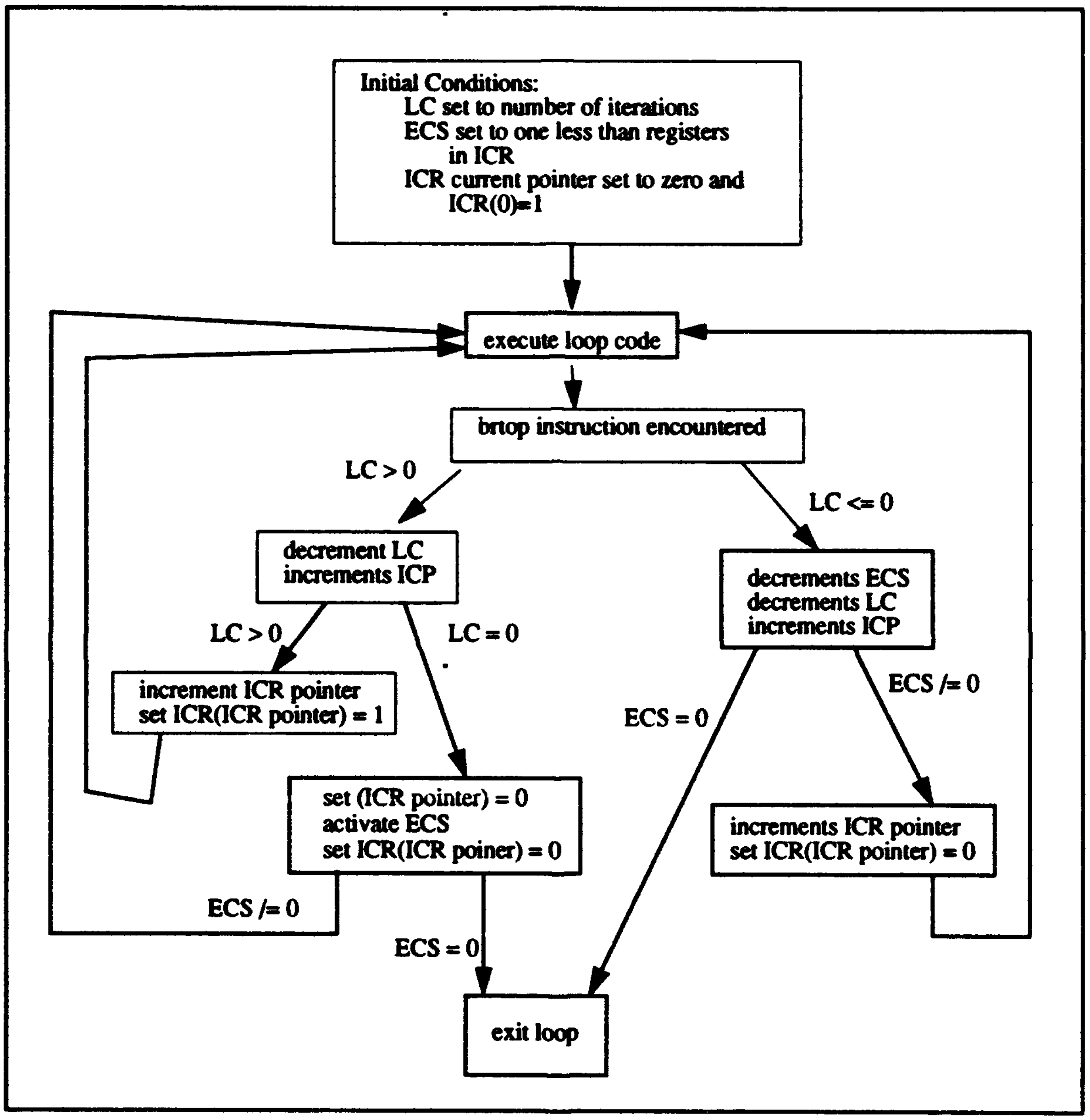
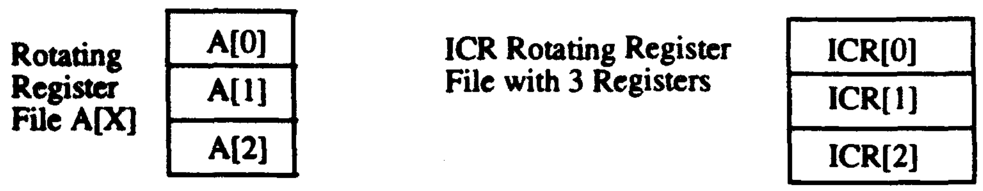

Initial Conditions:
LC set to number of iterations
ECS set to one less than registers in ICR
ICR current pointer set to zero and ICR(0)=1
execute loop code
brtop instruction encountered
LC > 0
LC <= 0
decrement LC increments ICP
decrements ECS decrements LC increments ICP
LC > 0
LC = 0
increment ICR pointer set ICR(ICR pointer) = 1
ECS = 0
ECS /= 0
set (ICR pointer) = 0 activate ECS set ICR(ICR poiner) = 0
increments ICR pointer set ICR(ICR pointer) = 0
ECS /= 0
ECS = 0
exit loop

Figure 25: Iteration Execution Control Flow Chart


Rotating Register File A[X]
A[0]
A[1]
A[2]
ICR Rotating Register File with 3 Registers
ICR[0]
ICR[1]
ICR[2]

Code Segment

```
LOOP_TOP
     INST1:                     if ICR(1)
     INST2, using A(ICP)        if ICR(0)
     INST3:                     if ICR(2)
     brtop LOOP_TOP
```

Assume that initially, the LC=4, the ECS=2. The following sequence occurs:

at top of iteration number	ICP	LC	ICR pointer	ECS	ICR(0) ICR(1) ICR(2)	instructions to be executed this iteration
1	0	4	0	2	ICR(0)=1 ICR(1)=0 ICR(2)=0	INST2with A(0)
2	1	3	1	2	ICR(0)=1 ICR(1)=1 ICR(2)=0	INST2 with A(1) and INST1
3	2	2	2	2	ICR(0)=1 ICR(1)=1 ICR(2)=1	INST2with A(2) INST1, INST3
4	3	1	3	2	ICR(0)=1 ICR(1)=1 ICR(2)=1	INST2with A(0) INST1, INST3
5	4	0	0	2	ICR(0)=0 ICR(1)=1 ICR(2)=1	INST1, INST3
6	5	-1	1	1	ICR(0)=0 ICR(1)=0 ICR(2)=1	INST3

Figure 26: Hardware Support Sequence of Events

To support the use of the hardware for loop pipelining, additional special instructions are made available. The initial specification of the register file namings, numbers, and size is obviously required. This can (and should) be done outside of the loop structure to minimize the overhead of specifying the requirements. It will be assumed that the register file requirements specific to the loop will be associated with a label (referred to as the "SET_UP_LABEL") which uniquely identifies the set up requirements. The information contained in these specifications is determined from the evaluations which will be discussed in Section IV.B.2 and Section IV.B.3.

In order to activate the specific requirements, special instructions are provided to initiate the use of the specifications. One instruction will be called the "SET_UP" instructions. This instruction takes as arguments the value that should be assigned to the LC, and the value that should be initially assigned to the ESC. This instruction should obviously be used prior to entering the pipelined kernel schedule. To initiate the use of the specifications associated with a labelled set up condition, a "INIT" is used, with the argument being the label of the set up specifications. This instruction will initialize the required register files set up for the specific used desired. The instruction should also be executed prior to the commencement of the pipelined kernel schedule. For the example shown in Figure 26, if the set up requirements were contained in specifications labeled "example', then the code sequence would be as follows:

```
SET_UP 4, 2
INIT example
LOOP_TOP
    INST1:                   if ICR(1)
    INST2, using A(ICP)      if ICR(0)
    INST3:                   if ICR(2)
    brtop LOOP_TOP
```

B. ISSUES OF CONCERN FOR CODE GENERATION

Some of the issues which must be considered to properly generate the final code structure after modulo scheduling are specifically addressed by Rau, Schlansker, and

Tirumalai [Ref. 6]. There are basically four issues that are required to be addressed to generate code after applying the scheduling technique presented in this thesis:

- adding loop control and loop control variable incrementation to the modified transformed DDG
- creating the final pipelined kernel schedule to be used as the new innermost loop code
- creating the prolog and epilog for the pipelined kernel schedule
- determining the transitioning areas of the iteration space where the pipelined kernel schedule cannot be applied
- determining the required amount of preconditioning of the inner loop before use the pipelined schedule.

Of these four issues, only the fourth is specific to the overall technique proposed by this thesis. The other four, however, are necessary to any Modulo Scheduling technique. The second, third, and fifth items are fully discussed in the previous work of Lam [Ref. 2], and Rau, Schlansker, and Tirumalai [Ref. 6]. The first item is seldom discussed, but practically is a concern and should be addressed. In any case, summaries of each of the issues are given in this section. In each discussion, the first situation considered is that of no special hardware support as described in Section IV.A.2, followed by the discussion of the simplifications allowed when the added support is available.

1. Adding Loop Control To The Modified Transformed DDG

In reality, the innermost loop code not only includes the loop body, but also the loop variable control instructions. Without hardware support, the instructions consist of an increment, some sort of a comparison, and a branch instruction. It is obviously beneficial if these instructions can be incorporated into the loop pipelining effort rather than merely be sequentially executed. This section, therefore, discusses the changes needed to be made to the modified transformed DDG to incorporate the control instructions into the scheduling procedure.

a. *Adding Loop Control Instructions With Basic Machine Support*

With only the basic VLIW machine support, the final modified transformed DDG for the innermost loop that will be used to generate the pipelined schedule will be slightly altered from the modified transformed one previously discussed (as shown in Figure 19 for the example). The alteration is simply that the loop control code for the innermost index variable is added to the modified transformed DDG in order to be included in the pipelined schedule. This control code basically consists of 3 instructions: an innermost loop variable increment instruction, an innermost loop ending comparison instruction, and a branch instruction for restarting the innermost loop when necessary (these instructions will be labeled S15, S16, and S17 in the example). The addition of this code is required independent the transformation method used or pipelining method used. It is desirable to include these instructions in the pipelining procedure, and the following discussion explains how this should occur.

(1) Adding Loop Control Code To The Loop Structure. For the example being pursued, the three code instructions which occur at the end of the innermost loop body will be labelled S15, S16, and S17. Additional registers required are R17 and R18, with R17 holding the value of the ending condition for the innermost loop. We will assume that the comparison instruction, S16, used requires the use of the ADDER. Additionally, the innermost loop label is added, used for the branch instructions branch location.

The resultant loop code is as follows:

```
for i'_1 in 3..700 loop
    calculate $R1 = max(1, \left\lceil \frac{i'_1 - 500}{2} \right\rceil)$
    calculate $R17 = min(\left\lfloor \frac{i'_1 - 1}{2} \right\rfloor, 100) + 1$
    LOOP2:
        S13(i'_2)    MULT R16, #2, R1
        S14(i'_1):   SUB R2, R15, R16
        S1(i'_2):    MULT R4, R3, R1
        S2(i_2):     SUB R5, R2, #1
        S3:          ADD R6, R4, R5
        S4:          LD R7, R6(R0)
        S5(i'_2):    SUB R8, R1, #1
        S6:          MULT R9, R3, R8
```

```
S7:          ADD R10, R2, #1
S8:          ADD R11, R9, R10
S9:          LD R12, R11(R0)
S10:         ADD R13, R7, R12
S11(i2):     ADD R14, R4, R2
S12:         ST R13(R0), R14
S15(i'2):    ADD R1, R1, #1
S16(i'2):    SGT R18, R1, R17
S17:         BNEZ R18, LOOP2
```

where, again:

- The register R0 is used as the base register for the array A(i,j).
- The register R1 is used to store the value of the i'_2 variable.
- The register R2 is used to store the value for the i_2 variable.
- The register R3 is used to store the length of each row, in this original case, this would have the value of 500. Other registers are assigned as necessary to complete the calculation.
- The register R15 is used to store the value of the i'_1 variable.
- The register R17 is used to store the calculated value for the stopping condition of the innermost loop
- LOOP2 is the label used to identify the beginning of the innermost loop

The starting and stopping values for the innermost loop control variable are calculated prior to the start of the loop as indicated above, and is not included as part of the innermost loop code

(2) Adding The Loop Control Code Nodes To The Modified Transformed DDG. Although inclusion of the code required for loop control in the loop The control instructions form a subgraph shown in Figure 27 which must be added to the modified transformed DDG.

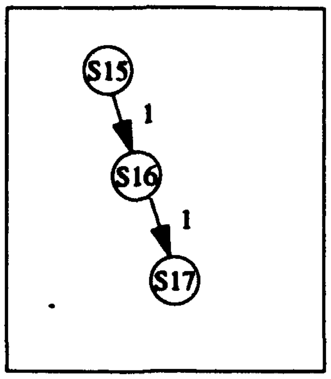


Figure 27: Subgraph For Loop Control Instructions

The subgraph is added to the DDG in a similar manner as were the transformation instructions. Because S15 defines a value for the loop control variable, there is a dependence between this node and any use of the control variable in the next iteration. The branch instruction causes a control dependence between itself and all nodes in the next iteration. By ensuring that a dependence arc is included between this branch and all of the nodes in the modified transformed DDG, no instructions from subsequent iterations will be executed unless the branch condition determines that additional execution of the loop is required. In that way, no mending will be required to fix inappropriately executed instructions from iterations which should not have occurred. The representation of this dependence can be simplified merely by ensuring a dependence arc exists between the branch and the topological "top" of the DDG.The resultant DDG for the innermost loop is shown in Figure 28.

The dependence arcs which extend from the branch instructions have a delay of "one", signifying that instructions are dependent on the previous iterations branch. Most of the dependences from the increment instruction, S15, are loop carried dependences (also having a loop delay of one).

Significant to note is that a simple cycle is introduced by instruction S15. The Acyclic DDG Modulo Scheduling technique was not intended to handle cycles. However, this simple cycle adds the constraint that there must be a latency of 1 between the

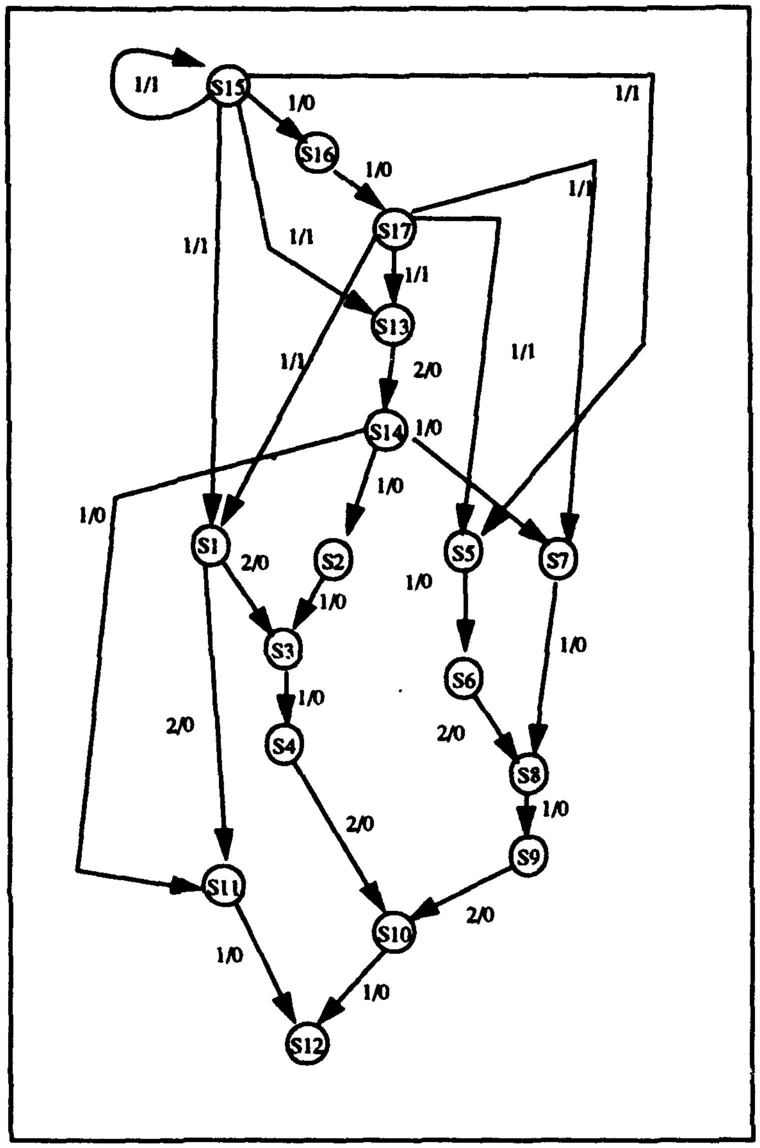


Figure 28: Final Innermost Loop DDG with Loop Control Code Added When There Is Basic Machine Hardware Support

execution of instruction S15 of two different iterations. By the nature of the scheduling process, any one instruction is only scheduled once in the Modulo Resource Reservation Table. Because the table must have at least one time slot, the constraint is trivially met, and will not cause a problem. With the addition of this code, we are otherwise guaranteed that no other cycle can be created. This is true because the modified transformed DDG is itself acyclic, and no instruction from this DDG can alter the input values to the control code instructions--that is, there can be no dependence arc back to the control code nodes to cause a cycle.

(3) The New Modulo Resource Reservation Table. With the addition of the loop control code to the modified transformed DDG, the Modulo Resource Reservation Table is generated as previously discussed. Assuming that there are two adders, one multiplier, one load/store, and now one branch unit available on the VLIW machine, the Acyclic DDG Modulo Scheduling technique is performed on the final modified transformed DDG of Figure 28, ignoring the simple cycle. The result is the Modulo Resource Reservation Table of Figure 29.

	Resource Unit				
time	adder	adder	multiplier	Load/Store	Branch
$5(k-a)+t_0$	$(S15)_k$	$(S5)_k$	$(S13)_k$	$(S4)_{k-1}$	
$5(k-a)+t_0+1$	$(S16)_k$	$(S10)_{k-2}$	$(S1)_k$	$(S9)_{k-1}$	
$5(k-a)+t_0+2$	$(S14)_k$	$(S11)_{k-1}$	$(S6)_k$	$(S12)_{k-2}$	
$5(k-a)+t_0+3$	$(S2)_k$	$(S7)_k$			
$5(k-a)+t_0+4$	$(S3)_k$	$(S8)_k$			$(S17)_k$

Figure 29: Final Modulo Resource Reservation Table With Basic Machine Hardware Support

The calculated IIII for generating the reservation table has now increased to five time units vice four, due to the addition of the control instructions of the resource requirements. The branch instruction, S17, has been placed in the last time slot of the schedule to control the jumping back to the top of the pipelined kernel.

b. Adding Loop Control Instructions With Special Machine Support

With the special machine support as describe in Section IV.A.2, much of the loop control for the innermost loop can be handled by the hardware. However, there is still a need to maintain the value of the index variable for referencing in the code. In addition, the branch instruction "brtop" will be needed to be scheduled as well. As a result, added to the modified transformed DDG will be an innermost loop variable increment instruction and the "brtop" instruction. The existence of the hardware also requires added instructions of SET_UP and INIT. These instructions must be added to the code just prior to using the pipelined loop. The placement of these instructions, however, will be discussed in Section IV.C.

(1) Adding Loop Control Code To The Loop Structure. The instructions S15 and S16 are added at the end of the loop. For the example being pursued, the two added instructions can be considered to occur at the end of the innermost loop body as they were in the previous case. The instructions will be labelled S15 and S16. The additional register R17 is again added to holds the ending value for the innermost loop variable.

The resultant loop code is as follows:

```
for i'_1 in 3..700 loop
    calculate $R1 = max(1, \left\lceil \frac{i'_1 - 500}{2} \right\rceil)$
    calculate $R_{end} = min(\left\lfloor \frac{i'_1 - 1}{2} \right\rfloor, 100) + 1$
    LOOP_TOP:
        S13(i'_2)     MULT R16, #2, R1
        S14(i'_1):    SUB R2, R15, R16
        S1(i'_2):     MULT R4, R3, R1
        S2(i_2):      SUB R5, R2, #1
        S3:           ADD R6, R4, R5
```

```
S4:         LD R7, R6(R0)
S5(i'2):    SUB R8, R1, #1
S6:         MULT R9, R3, R8
S7:         ADD R10, R2, #1
S8:         ADD R11, R9, R10
S9:         LD R12, R11(R0)
S10:        ADD R13, R7, R12
S11(i2):    ADD R14, R4, R2
S12:        ST R13(R0), R14
S15(i'2):   ADD R1, R1, #1
S16:        BRTOP LOOP_TOP
```

where, again:

- The register R0 is used as the base register for the array A(i,j).
- The register R1 is used to store the value of the i'_2 variable.
- The register R2 is used to store the value for the i_2 variable.
- The register R3 is used to store the length of each row, in this original case, this would have the value of 500. Other registers are assigned as necessary to complete the calculation.
- The register R15 is used to store the value of the i'_1 variable.
- The register R_{end} is used to store the calculated value for the stopping condition of the innermost loop. This stopping condition is not explicitly needed for loop control, but will be used to calculate the number of innermost loop iterations. An actual register number (R14) will be assigned to this calculated value in the code generation process to be discussed later.
- LOOP_TOP is the label used to identify the beginning of the innermost loop.

The starting and stopping values for the innermost loop control variable are calculated prior to the start of the loop as indicated above, and is not included as part of the innermost loop code.

(2) Adding The Loop Control Code Nodes To The Modified Transformed DDG. The control instructions in this case are added to the modified transformed DDG as they were in the case of no hardware support. However, this time the increment node and

the branch node are not dependent upon each other. The resultant modified transformed DDG for the innermost loop is shown in Figure 30.

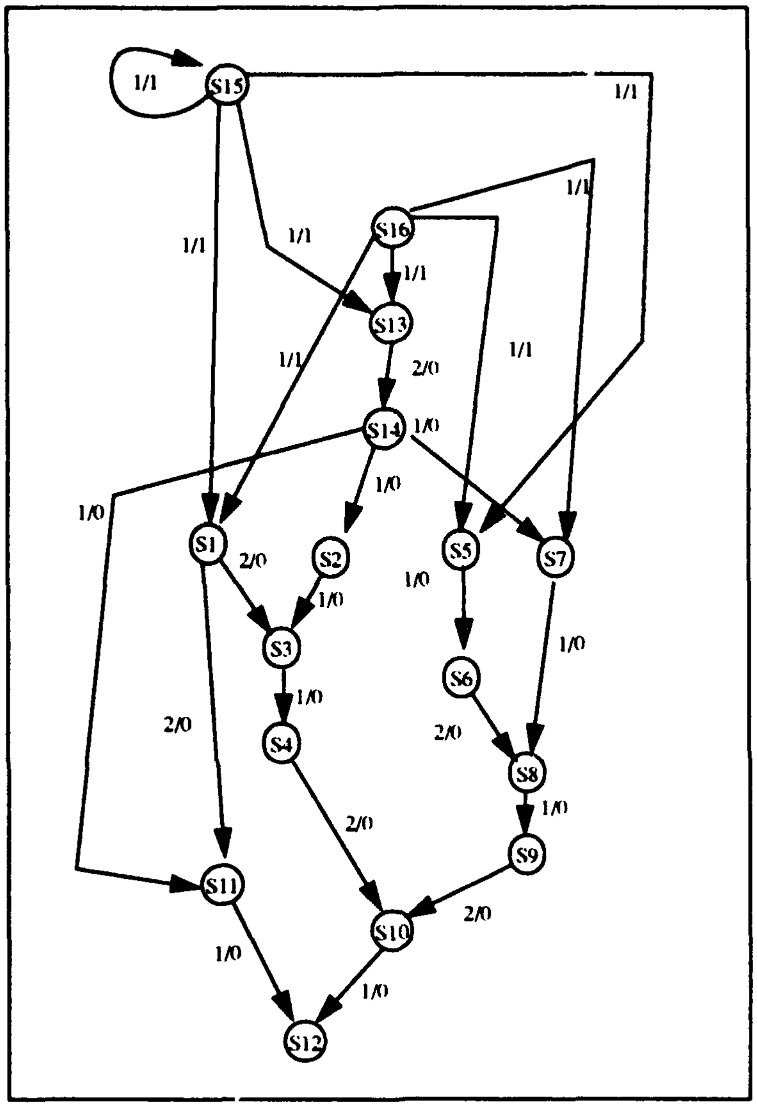


Figure 30: Final Innermost Loop DDG with Loop Control Code Added When There Is Special Machine Hardware Support

(3) The New Modulo Resource Reservation Table. Re-performing the Acyclic DDG Modulo Scheduling Procedure on the modified transformed DDG of Figure 30, the result is the Modulo Resource Reservation Table of Figure 31.

	Resource Unit				
time	adder	adder	multiplier	Load/Store	Branch
$5(k-a)+t_0$	$(S15)_k$	$(S5)_k$	$(S13)_k$	$(S4)_{k-1}$	
$5(k-a)+t_0+1$	$(S10)_{k-2}$	$(S11)_{k-1}$	$(S1)_k$	$(S9)_{k-1}$	
$5(k-a)+t_0+2$	$(S14)_k$		$(S6)_k$	$(S12)_{k-2}$	
$5(k-a)+t_0+3$	$(S2)_k$	$(S7)_k$			
$5(k-a)+t_0+4$	$(S3)_k$	$(S8)_k$			$(S16)_k$

Figure 31: Final Modulo Resource Reservation Table With Special Machine Hardware Support

2. Creating The Final Pipelined Kernel Schedule

Once the Modulo Resource Reservation Table has been generated, the final pipelined kernel schedule which is used as the new inner loop code can be derived. This pipelined kernel schedule is basically created directly from the reservation table. The only complication that may exist occurs when explicit specification of register usage is required, as with the ongoing example. When this is the case, the overlapping of different iterations in a software pipelined inner loop may also create a problem with register usage overlap.

The problem can be explained using an example from Lam [Ref. 2]. Assume a loop code fragment that uses the register R1 exists such as in the following:

```
S1: def(R1)
S2: operation
S3: use(R1)
```

With three general processors available, the Modulo Resource Reservation Table which would be produced would be that shown in Figure 32.

time from beginning of loop	Processor P1	Processor P2	Processor P3
0	$(S3)_{k-2}$	$(S2)_{k-1}$	$(S1)_k$

Figure 32: Modulo Resource Reservation Table

Using the Modulo Resource Reservation Table of Figure 32 to construct the pipelined loop body would result in an execution timing diagram as shown in Figure 33, with an IIII of one time unit and the kernel first being used at time 2. In this figure, the statement labels have been replaced by the actual instructions to better illustrate the problem.

time	iteration number 1	2	3	4	5
0	def(R1)				
1	operation	def(R1)			
2	use(R1)	operation	def(R1)		
3		use(R1)	operation	def(R1)	
4			use(R1)	operation	def(R1)
5				use(R1)	operation
6					use(R1)
7					
8					

. . .

possible kernel for pipelined loop, with IIII of one time unit (row 2)

.
.
.

Figure 33: Initial Timing Table For Pipelined Iterations

Because of the explicit assignment of registers, a register usage anti-dependence (a dependence that normally requires a variable usage prior to a later variable definition) is frequently created which is dependent upon the use of registers and not on the actual data. For the above example, the use of the register R1 in one iteration occurs after the redefinition of R1 in the next iteration. This will result in the use of the wrong data value in R1. To alleviate this problem, IIII could be extended to two time units, but this reduces the efficiency of the pipelined schedule created. Better solutions to this problem depend upon the support given by the hardware, but in all cases, some register renaming scheme is followed to avoid rewriting to registers prior to their proper usage.

a. Renaming Registers With Basic Machine Support

A technique which Lam [Ref. 2] labelled *Modulo Variable Expansion* is employed to solve the register renaming problem when there is only the basic machine hardware support. Modulo Variable Expansion requires repetition of the schedule generated by the Modulo Resource Reservation Table, and explicit renaming of the registers in the appropriate instructions to ensure there is no loss of information. For the simple example given above, the result would require the renaming of the R1 register in every other iteration, yielding an the timing diagram shown in Figure 34. The IIII will remain one time unit in this case, but the pipelined loop has been unrolled to include two iterations. The timing diagram of Figure 34 has the unrolled pipelined kernel in Figure 35.

To conduct Modulo Variable Expansion, the usage lifetimes of each register definition must be evaluated. This determines the number of needed namings (i.e., the number of different registers) of the register in order to avoid overwriting a register before the information it contains can be used.

	iteration number				
time	1	2	3	4	5
0	def(R1)				
1	operation	def(R2)			
2	use(R1)	operation	def(R1)		
3		use(R2)	operation	def(R2)	
4			use(R1)	operation	def(R1)
5				use(R2)	operation
6					use(R1)
7					
8					

. . .

pipelined loop kernel still has IIII of 1 time unit, but loop has been unrolled so that there are two iterations per pipelined body.

.
.
.

Figure 34: Table For Pipelined Iterations with Modulo Variable Expansion Applied

time from beginning of loop	Processor		
	P1	P2	P3
0	use(R1)	operation	def(R1)
1	use (R2)	operation	def(R2)

Figure 35: Pipelined Kernel with Modulo Variable Expansion Applied

The number of renamings is given by the equation:

$$N_{namings\ of\ Reg_r} = \left\lceil \frac{Lifetime_r}{IIII} \right\rceil \qquad \text{(Eq. 10)}$$

where Reg_r is a register

Each renaming of a register occurs in a different copy of the reservation table copy. Because differenct registers may need to be renamed a different number of times, the reservation table schedule must be repeated an appropriate number of times to accommodate all of the registers. The required number of repetitions of the reservation table schedule is therefore determined by the equation:

$$N_{schedule\ repetitions} = least\ common\ multiple\ [N_{namings\ of\ Reg_r}], \quad for\ all\ Registers\ Reg_r\ used \qquad \text{(Eq. 11)}$$

b. Register Renaming With Special Machine Support

Special machine hardware supported solutions revolve around use of the Rotating Register Files. A rotating register file is created for each of the originally addressed registers which require renaming. The number of renamings can be determined as in the above discussion, but use of the RRF will eliminate the need to unroll the pipelined loop and duplicate code.

For the simple example above, a rotating register file would be created for the R1 register, consisting of two registers, R1[0] and R1[1]. The resultant timing diagram is shown in Figure 36 with the pipelined kernel schedule shown in Figure 37. In these diagrams, the current ICP value modulo 2 is used to determine the appropriate rotating register file register that is to be referenced. With the ICP starting at 0, the timing table generated using the hardware support is precisely the schedule with R1 being replaced by R1[0] and R2 being replaced by R1[1].

	iteration number				
time	1	2	3	4	5
0	def(R1[ICP])				
1	operation	def(R1[ICP])			
2	use(R1[ICP])	operation	def(R1[ICP])		
3		use(R1[ICP])	operation	def(R1[ICP])	
4			use(R1[ICP])	operation	def(R1[ICP])
5				use(R1[ICP])	operation
6					use(R1[ICP])
7					
8					

. . .

X represents the current reference pointer to the register file R1. The IIII is still one time unit

.
.
.

Figure 36: Timing Table for Pipelined Iterations with Rotating Register File Support

	Processor		
time from beginning of loop	P1	P2	P3
0	use(R1[ICP])	operation	def(R1[ICP])

Figure 37: Pipelined Kernel with Rotating Register File Support

c. *The Original Example*

Returning to the example which produced the Modulo Resource Reservation Tables of Figure 29 and Figure 31, the pipelined kernel schedule can be created in either the case of without or with special hardware support.

(1) Creating The Pipeline Kernel Schedule With Basic Hardware Support. First assume that there is only the basic hardware support to solve the register remaning problem. The lifetime analysis indicates that registers R1, R4 and R7 have a lifetime of between five and eleven time units, and all other registers have a lifetime of five time units or less. Hence, the value of N_{namings} is two for R1, R4 and R7, resulting in the value of $N_{\text{schedule repetitions}}$ also being two.

For convenience, the renamed registers for R1, R4, and R7 will be referred to as R1[0] and R1[1] for R1, R4[0] and R4[1] for R4, and R7[0] and R7[1] for R7. The resulting pipelined kernel is then given by Figure 38.

As can be seen, the schedule from the reservation table is repeated twice. Those registers that required more than one name are included with the associative statement in which the registers are used, with appropriate index numbering identifying the proper renamed register. Registers which require only one name are not indicated. Important to note, only one control branch instruction is included in this schedule, to ensure that the kernel is executed at the end of the kernel schedule, and not in the middle. This will become important for the generation of transition code discussed in the next section.

(2) Creating The Pipeline Kernel Schedule With Special Hardware Support. Assume that the hardware support of rotating register files is available for use in solving the register renaming problem. The use of hardware support both eliminates one instruction that must be scheduled as well as the dependences associated with that node. As a result, the lifetime analysis indicates that registers R1, R7, and R14 require renaming. However, with the added support of the RRF and ICP, then explicit repetitions of the Modulo Resource Reservation Table is unnecessary to create the pipelined kernel schedule.

	Resource Unit				
time	adder	adder	multiplier	Load/Store	Branch
0	S15 uses R1[0], defined R1[1]	S5 uses R1[0]	S13 uses R1[0]	S4 defined R7[1]	
1	S16 uses R1[1]	S10 use R7[0]	S1 defined R4[0] uses R1[0]	S9	
2	S14	S11 uses R4[1]	S6	S12	
3	S2	S7			
4	S3	S8			
5	S15 uses R1[1], defined R1[0]	S5 uses R1[1]	S13 uses R1[1]	S4 defined R7[0]	
6	S16 uses R1[0]	S10 use R7[1]	S1 defined R4[1] uses R1[1]	S9	
7	S14	S11 uses R4[0]	S6	S12	
8	S2	S7			
9	S3	S8			S17

Figure 38: Final Pipelined Kernel Schedule with Modulo Variable Expansion and Basic Machine Hardware Support

Again let a register file of two registers be established for each of registers R1, R4, and R7,with the register files can be referred to as R1[(X1)] for R1, R7[(X7)] for R7,a nd R14[(X14)] for R14. The variable (X1) refers to the referencing pointer use to access the registers R1[0] and R1[1]. Variables X7 and X14 perform similar functions with their respective register files. In any iteration, these variables can be functions of the current value of ICP. The variables X1, X7, and X14 are evaluated modulo the number of registers in each respective register file (in each case modulo 2) in order to reference the registers on a rotating basis. The pointer values are initialized to zero at the beginning of the loop by the "INIT" instruction and are incremented automatically at the start of each new kernel execution. The resulting pipelined kernel is then given by Figure 39.

	Resource Unit				
time	adder	adder	multiplier	Load/Store	Branch
0	S15 uses R1[ICP], defined R1[ICP+1]	S5 uses R1[ICP]	S13 uses R1[ICP]	S4 defined R7[ICP+1]	
1	S10 use R7[ICP]	S11 defined R14[ICP+1]	S1 uses R1[ICP]	S9	
2	S14		S6	S12 use R14[ICP]	
3	S2	S7			
4	S3	S8			S17

Figure 39: Final Pipelined Kernel Schedule with Special Hardware Register Renaming Support

The schedule from the reservation table is mirrored exactly, with proper pointer references indicating the proper relationship between register definitions and uses. References to the register file R1[X1] are included with the associated statement in which the registers are used, while registers which require only one name are not indicated.

3. Creating The Prolog And Epilog For The Pipelined Kernel Schedule

Once the pipelined kernel schedule has been created, the next consideration in code generation is the creating the code segments which provide the needed transition to the pipelined loop. These code segments are called the *prolog* and the *epilog*, and are created from partial inner loop schedules (actually, partial Modulo Resource Reservation Table schedules), and allow the starting and completing of iterations which are only partially represented at the beginning and end of the pipelined loop body.

The prolog supplies the front end transition into the pipelined loop, and the epilog provides the transition at the end of the pipelined loop execution. If the instructions in the Modulo Resource Reservation Table spanned across N_{alive} different iterations, then there will be $(N_{alive}$-1) partial schedules in both the prolog and the epilog. The first partial schedule of the prolog will be the one which consists only of those instructions that are "latest" (i.e., those with the highest statement index k+1, k, k-1, etc.) in the Modulo Resource Reservation Table. The second partial schedule will include these instructions as well as the instructions that are second "latest", and so on, until all but the "earliest" instructions are included. These "earliest" instructions are first executed in the pipelined kernel schedule.

The epilog partial schedules are similarly pattern. The first partial schedule consists of all instructions except for the "latest" as indicated in the Modulo Resource Reservation Table, with each subsequent partial schedule eliminating the next latest set of instructions. The last partial schedule of the epilog includes only the "earliest" reservation table instructions.

In all partial iterations, the loop control branch instruction is not included in the scheduling.

a. Creating The Prolog And Epilog With Basic Machine Support

Without only the basic machine hardware support, the prolog and epilog must be determined explicitly and be included as transition code into the pipelined kernel body. The register renaming scheme used to create the kernel must also be extended into these regions to ensure that the proper register referencing is maintained.

b. Creating The Prolog And Epilog With Special Machine Support

Special machine hardware support can again be used to aid in the creation of the prolog and epilog. The explicit determination of the prolog and epilog required with basic machine support can be avoided by using the Iteration Control Register.

A single instruction group is made up of all of the instructions of the Modulo Resource Reservation Table which has the same iteration index identifier. One register in the ICR identifies if the instructions of a group in the kernel should or should not be executed during a given iteration. Only during the prolog or epilog will any instruction have a negative predicate and not be executed.

With this special hardware support available, the prolog and epilog are generated from the pipelined kernel schedule during run time. Initially, the SET_UP instruction is used to set all predicates except the first (p_0) are false, set LC to the number of iterations that must be executed, set the current ICR pointer to the first predicate register (p_0), set the first predicate register value to true (one), and set the ESC to the value of (N_{alive}-1). Each of the instructions in the kernel schedule is assigned a predicate register based on their relative iteration index, so that an instruction with iteration index of (k-x) is assigned the predicate register p_x, and is executed "if ICR(x)". The only instruction which in an exception to this is the brtop which will always have a true predicate, and is therefore always executed.

As described before, with the execution of the brtop instruction, counters are adjusted appropriately and the current ICR pointer moves to the next register. If the LC is greater than zero, the new current predicate is set to true. If the LC is now zero or less, the predicate is set to false, and the ESC counter is decremented. The partial kernel schedules are executed until the LC an the ESC are zero.

In this way, the execution sequence progressively adds instructions groups until the steady state kernel is reached. This performs the same function as a prolog which was explicitly generated before. The epilog is dynamically created by eliminating additional instruction groups from successive kernel repetitions until all instruction group predicates are negative, essentially draining the loop pipeline and completing the execution of the final iterations.

This "kernel only" execution requires the use of predicates and execution of the schedule a total of $[N_{inner} + (N_{alive} - 1)]$ vice N_{inner} repetitions. As explained in Section IV.A.2, the initialization of the counters is done with special initialization instruction "SET_UP" with arguments being the value of LC, ESC, and "set-up label". The instruction "INIT 'set-up label'" can be used to set the current ICR to the first register file and trigger the counters to take affect. The specifications for the ICR register file can be made prior to the loop execution at the same time that the specification requirements for the RRF were established and labeled.

c. *The Original Example*

Considering again the example with reservation tables of Figure 29 and Figure 31. The results of this step can be explained for both the case of no additional hardware support and the case of special hardware support.

(1) Creating The Prolog and Epilog With Basic Machine Support. In the case of a VLIW machine with basic hardware support, the prolog and epilog are generated using the Modulo Resource Reservation Table of Figure 29 with the renaming scheme utilized in Section IV.2.c. In this case N_{alive} is three, requiring that the prolog and epilog both have two partial iterations of the reservation table schedule. The prolog is shown in Figure 40 and the epilog is shown in Figure 41.

time	Resource Unit				
	adder	adder	multiplier	Load/Store	Branch
0	S15 uses R1[0], defined R1[1]	S5 uses R1[0]	S13 uses R1[0]		
1	S16 uses R1[1]		S1 defined R4[0] uses R1[0]		
2	S14		S6		
3	S2	S7			
4	S3	S8			
5	S15 uses R1[1], defined R1[0]	S5 uses R1[1]	S13 uses R1[1]	S4 defined R7[0]	
6	S16 uses R1[0]		S1 defined R4[1] uses R1[1]	S9	
7	S14	S11 uses R4[0]	S6		
8	S2	S7			
9	S3	S8			

Figure 40: Prolog For Modulo Resource Reservation Table of Figure 29 and Pipelined Kernel Schedule of Figure 38

	Resource Unit				
time	adder	adder	multiplier	Load/Store	Branch
0				S4 defined R7[1]	
1		S10 use R7[0]		S9	
2		S11 uses R4[1]		S12	
3					
4					
5					
6		S10 use R7[1]			
7				S12	
8					
9					

Figure 41: Epilog For Modulo Resource Reservation Table of Figure 29 and Pipelined Kernel Schedule of Figure 38

(2) Note that the branching instruction is eliminated from both the prolog and epilog schedules because it is not part of the loop, but merely a transitioning section prior to the loop.

As can be seen, the first partial iteration of the prolog includes all those instructions from the Modulo Resource Reservation Table with the latest iteration index. Subsequent partial iterations sequentially add the instructions with the next latest iteration index until the first complete schedule can be executed in the pipelined kernel. One complication that arises due to the Modulo Variable Expansion is the need to ensure that the correct renamed registers are initialized with the correct value. In the example, the register R1[0] is the first R1 register to be used, so it must hold the starting value of the variable represented by R1. The same requirement holds true for the register R7.

Likewise, the first partial iteration of the epilog includes all those instructions from the Modulo Resource Reservation Table except for those with the latest iteration index. Subsequent partial iterations sequentially eliminate the instructions with the next latest iteration index until all instructions have been eliminated from the inclusion. In both the prolog and epilog, the branch instructions are eliminated from the code.

(3) Creating The Prolog and Epilog With Special Hardware Support. With the special machine support of an *Iteration Control Register*, the instructions can be divided into three predicate groups: those with iteration indices of k, (k-1), and (k-2). The instructions with iteration index of k will have as their predicate the statement "if ICR(0)", those with iteration index of (k-1) will have as their predicate the statement "if ICR(1)", and those with iteration index of (k-2) will have as their predicate the statement "if ICR(2)".

For the first execution of the pipelined kernel, the predicate for those instructions with iteration index of k should be true and all others false, allowing the execution of the correct instructions. The second execution of the pipelined kernel, should have the instructions with indices of k or (k-1) executed. For the rest of the iterations, all instructions are executed, until the last iteration has started. At this point, LC would be zero, and the ESC engages to help create the epilog. The execution of (N_{alive} -1), that is, two, more kernel schedules execute, the first of which executes only those instructions which have indices of (k-1) ~~or (k-2)~~, and the last executing only those instructions with index (k-2).

4. Areas Of The Iteration Space Not Supporting Use Of The Pipelined Loop

The third consideration in code generation is the determination of the sections of the new iteration space to which the pipelined kernel schedule cannot be applied. In general, these areas will be labelled the *iteration space transition areas*. As mentioned earlier, the existence of these transition areas is the only problem of code generation which is truly unique to the technique presented in this paper. The determination of these areas is needed, however, because of the wavefront transformation which was originally applied. This tranformation skewed the iteration space of the two innermost loops, resulting in an iteration space with a parallelogram shape (see Figure 42), where the number of iterations in the inner loop tapers up from one and down to one near the bounds of the outermost loop. The use of the pipelined kernel schedule, as well as the supporting prolog and epilog, is obviously possible only if the number of iterations in any transformed innermost loop accommodates at least the number of iterations required by the prolog and a single repetition of the pipelined kernel schedule. The tranformation, therefore, creates the need to consider the transition areas.

a. Iteration Space Transition Areas With Basic Machine Support

With only basic machine hardware support, the pipelined kernel generation may have required repetitions of the modulo schedule to support explicit register renaming via modulo variable expansion. In this case, to use the resultant pipelined kernel schedule, the number of inner loop iterations must meet the requirement:

$$N_{inner} \geq [N_{schedule\ repetitions} + (N_{alive} - 1)] \qquad \text{(Eq. 12)}$$

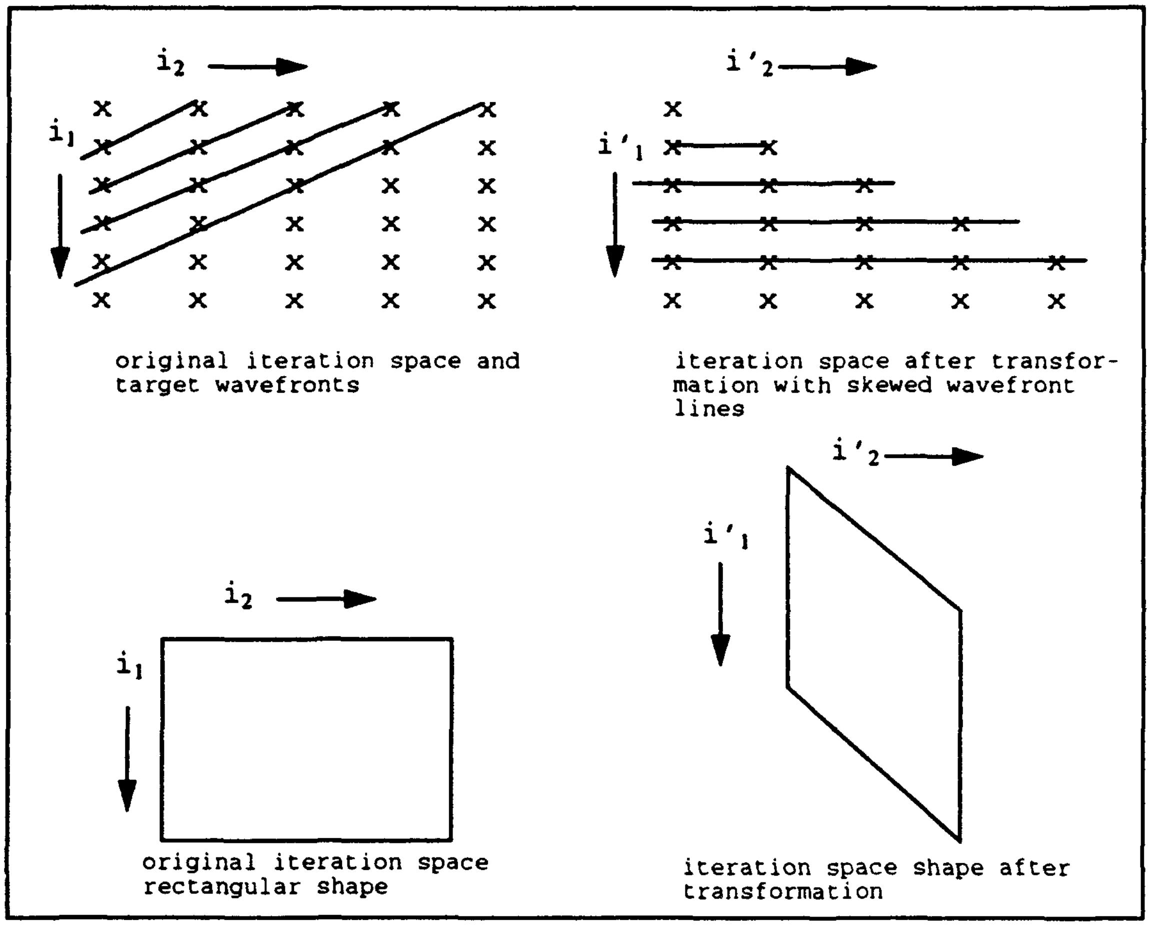

i_2
i_1
original iteration space and target wavefronts
i'_2
i'_1
iteration space after transformation with skewed wavefront lines
i'_2
i'_1
i_2
i_1
original iteration space rectangular shape
iteration space shape after transformation

Figure 42: Iteration Space Shape Characteristics, Before and After Transformation

For those second innermost iterations which do not have enough innermost iterations to meet the above requirement, the associated innermost loops should be executed without using the pipelined kernel schedule (in the worst case, these sections could be executed sequentially). Sections of non-pipelined iterations will therefore occur prior to, as well as following, the use of pipelined schedule. These sections of the iteration space are the iteration space transition areas

b. Iteration Space Transition Areas With Special Machine Support

With special machine hardware support, schedule repetitions were not needed to support register renaming, nor is explicit prolog and epilog generation required. As a result, the number of innermost loop iterations which are required to exist in order to support the use of the pipelined schedule is that amount which is represented in one schedule kernel. In this case, the value of $N_{schedule\ repetitions}$ in the above equation becomes one, hence, the number of inner loop iterations must merely meet the requirement:

$$N_{inner} \geq N_{alive} \qquad \text{(Eq. 13)}$$

Again, for those second innermost iterations which do not have enough innermost iterations to meet the above requirement, the associated innermost loops should be executed without using the pipelined schedule. As was the case when there is only basic hardware support, the iteration space transition areas of non-pipelined iterations will therefore occur prior to, as well as following, the use of pipelined kernel schedule

c. The Original Example

In the original example, the iteration space transition areas will differ depending on the machine hardware support.

(1) With Basic Machine Hardware Support. For the case of only basic machine hardware support, the value of $N_{schedule\ repetitions}$ in the example was two and the value of N_{alive} was three. As a result $[N_{schedule\ repetitions} + (N_{alive} - 1)]$ evaluates to four.

Therefore, the bounds on i'_1 for which the inner loop code cannot be executed using the pipelined schedule are:

for i'_1 in 3..8 and for i'_1 in 695..700

Hence, the areas defined by these boundaries become the iteration space transition areas.

(2) With Hardware Support. With hardware support, the value of $N_{schedule\ repetition}$ is essentially one, so that the innermost loop must have three iterations to use the pipelined kernel schedule. As a result, the bounds on i'_1 for which the inner loop code cannot be executed using the pipelined schedule are:

for i'_1 in 3..6 and for i'_1 in 697..700

As before, the areas defined by these boundaries become the iteration space transition areas.

5. Determination Of The Pipelined Loop Preconditioning

The last issue which must be addressed before presenting the code generation process is the determination of the amount of preconditioning that the innermost loop requires in order to use the pipelined kernel schedule. *Preconditioning* refers to the execution of non-pipelined iterations within a single execution of the innermost loop in order to allow the pipelined kernel schedule to execute the rest of the iterations. The need for preconditioning exists because the pipelined kernel schedule, when combined with the prolog iterations, can only execute a specific number of iterations.

To help understand this need, first consider the ideal case when no preconditioning would be required. This condition exists when it is known at the beginning of the innermost loop that the number of iterations in the inner loop which must be executed satisfies the equation:

$$N_{inner} = A \times N_{schedule\ repetitions} + (N_{alive} - 1) \qquad \text{(Eq. 14)}$$

where "A" is some positive integer.

From this equation, it can be seen that preconditioning is **only required in the case where no special machine hardware support was available** for implementing the register renaming required. When special hardware support is provided, then $N_{schedule\_repetitions}$ is essentially one, and the equation can be satisfied if $N_{inner} \geq N_{alive}$. However, from the previous section, this will always be the case when using the pipelined kernel schedule with special hardware support. Consequently, no preconditioning will ever be needed when using hardware support.

However, with only the basic machine hardware support for register renaming, this is unlikely that the equation is met in the general case for innermost loops, particularly because the skewing results in consecutive innermost loops having different numbers of iterations. Even if the transformed space is rectangular (i.e., no skewing was needed because the original loop was fully parallel), the number of innermost loop iterations still may not be such as to meet the above requirement. The solution is to identify the "extra" iterations of the innermost loop, which, after execution, will allow the remaining iterations of the innermost loop to satisfy the above equation. These "extra" iterations are executed without using the pipelined schedule (in a manner similar, perhaps, to the way in which the non-pipelined code required by Section IV.4), and are considered the *preconditioning iterations* for the pipelined loop.

Execution of the preconditioning iterations is then followed by the execution of the remaining iterations using the pipelined loop schedule. This preconditioning of the pipelined loop is described By Rau, Schlansker, and Tirumalai [Ref. 6]. The number of iterations required to be performed in the preconditioning code is given by the equation:

$$N_{precondition} = [N_{inner} - (N_{alive} - 1)] \; mod \; N_{schedule\ repetitions} \qquad \text{(Eq. 15)}$$

Again note that when register renaming machine hardware support is available, the value of $N_{schedule\ repetitions}$ is one, making $N_{precondition}$ equal to zero.

Because the skewing employed in the technique presented in this paper can result in differing numbers of iterations for different innermost loop executions, **the value of $N_{precondition}$ must be calculated following the start of each second innermost iteration**, and then immediately used to execute the preconditioning code for the subsequent innermost loop.

If Modulo Variable Expansion is used to rename registers in the pipelined schedule, then the appropriate registers (those that needed to be renamed) will need to be initialized as required before the prolog is commenced, to ensure that the correct values are used at the start of the prolog.

For the example, assuming only basic machined hardware support, the equation for $N_{precondition}$ is specifically:

$$N_{precondition} = [N_{inner} - 3] \, mod \, 2 = [N_{inner} - 1] \, mod \, 2 \qquad \text{(Eq. 16)}$$

C. GENERATING THE FINAL LOOP CODE

Now that the general issues concerning the generation of the final pipelined loop code structure have been completed, the actual code generation process can be described. As the previous sections demonstrated, the type of machine hardware support will affect the issues surrounding the code generation process. It can be concluded from these sections that use of special hardware support not only simplifies the creation of the pipelined kernel schedule, the prolog, and the epilog, but it also increases the potential benefit of using the pipelined kernel schedule to execute the iterations. This is true because it eliminates the schedule unrolling that might be required for Modulo Variable Expansion. Modulo Variable Expansion, in general, increases the size of the iteration space transition areas and the number of preconditioning iterations required to use the pipelined schedule, as explained in Section IV.B.4 and Section IV.B.5. These sections of code must be executed in a less efficient manner than the compact pipelined schedule (and in the worst case, sequentially), and hence reduce performance. The use of the special hardware support will minimize (or eliminate, in the case of preconditioning) these areas of code. **The approach to code generation, therefore, will assume that special machine hardware support is made available as previously described.**

Each of the considerations for code generation provides information which is used to create the final code structure. All of the information required can be obtained prior to loop execution, with the exception of the actual number of iteration within the innermost loop. This number is dependent on the second innermost loop control variable value. In any case, the information required for code generation has the identical form for any loop being pipelined. A general procedure can therefore be presented which will use the information to create the final code product.

1. Modelling The Final Loop Code Structure

In order to better describe and motivate the specific aspects of the code generation process, a loop code model is presented. The use of the model is intended to help organize and clarify the different segments of code which are required to be executed in a loop structure. The loop code structure model presented is, in a sense, a dependence diagram which identifies coarser-grained components than individual instructions.

a. Modelling The Original Loop Structure Code

To introduce the loop structure code model, the model will be applied to the original loop structure code. The original code loop was in the form:

```
for $i_1$ in 1..$N_1$ loop
    for $i_2$ in 1..$N_2$ loop
                .
                .
                .
            for $i_{n-1}$ in 1..$N_{n-1}$ loop
                for $i_n$ in 1..$N_n$ loop
                    (original loop body)
                end loop
            end loop
                .
                .
                .
    end loop
end loop
```

This code segment can be modelled by a code segment dependency diagram that illustrates the sequence of instructions required for this loop, as shown in Figure 43. The diagram simplifies the code segment by recursively defining the "subloop 2 → n" node as per Figure 44. Figure 44.a and Figure 44.b indicate the different subloop structures depending on whether or not the subloop is the innermost loop.

In the diagrams, the nodes represent segments of code with a specific function. Arcs represent data dependencies and flow dependencies, with the data dependencies shown with solid lines and the flow dependencies shown with dashed lines. If the flow change was due to a test and branch requirement, the arcs are labelled with a "BT" or a "BF", indicating a branch when condition is true or a branch when condition is false, respectively.

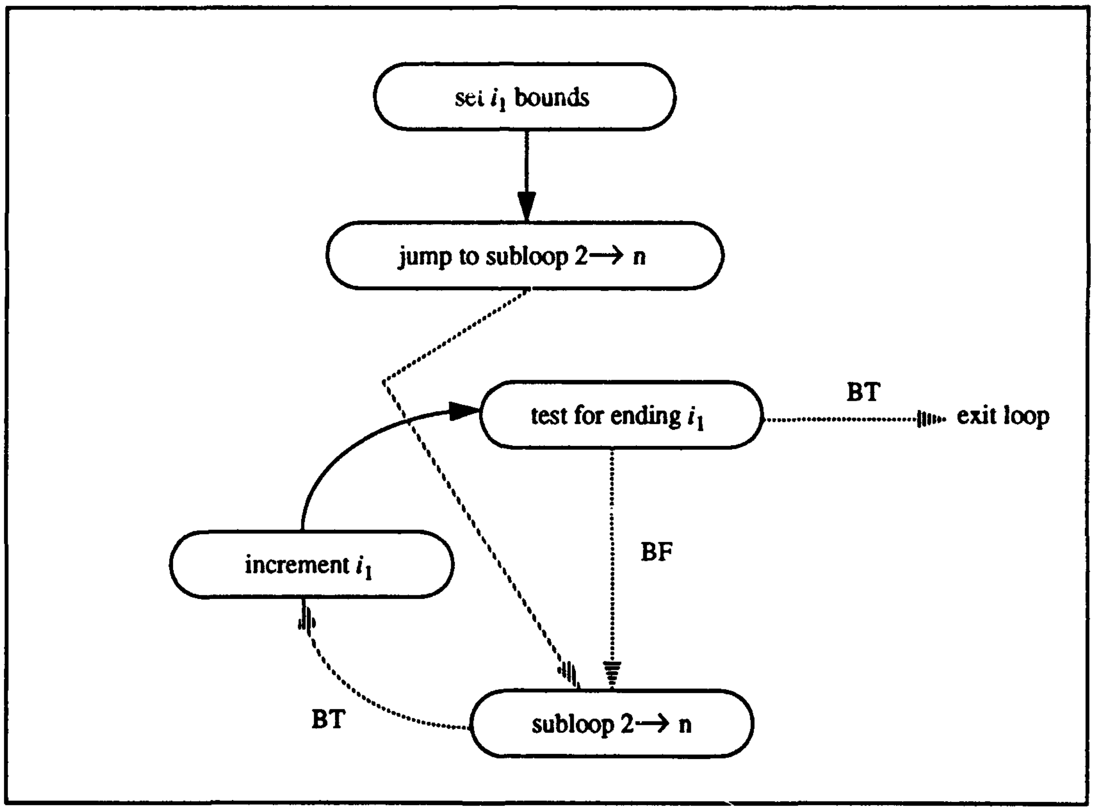


Figure 43: Original Loop Structure Code Model

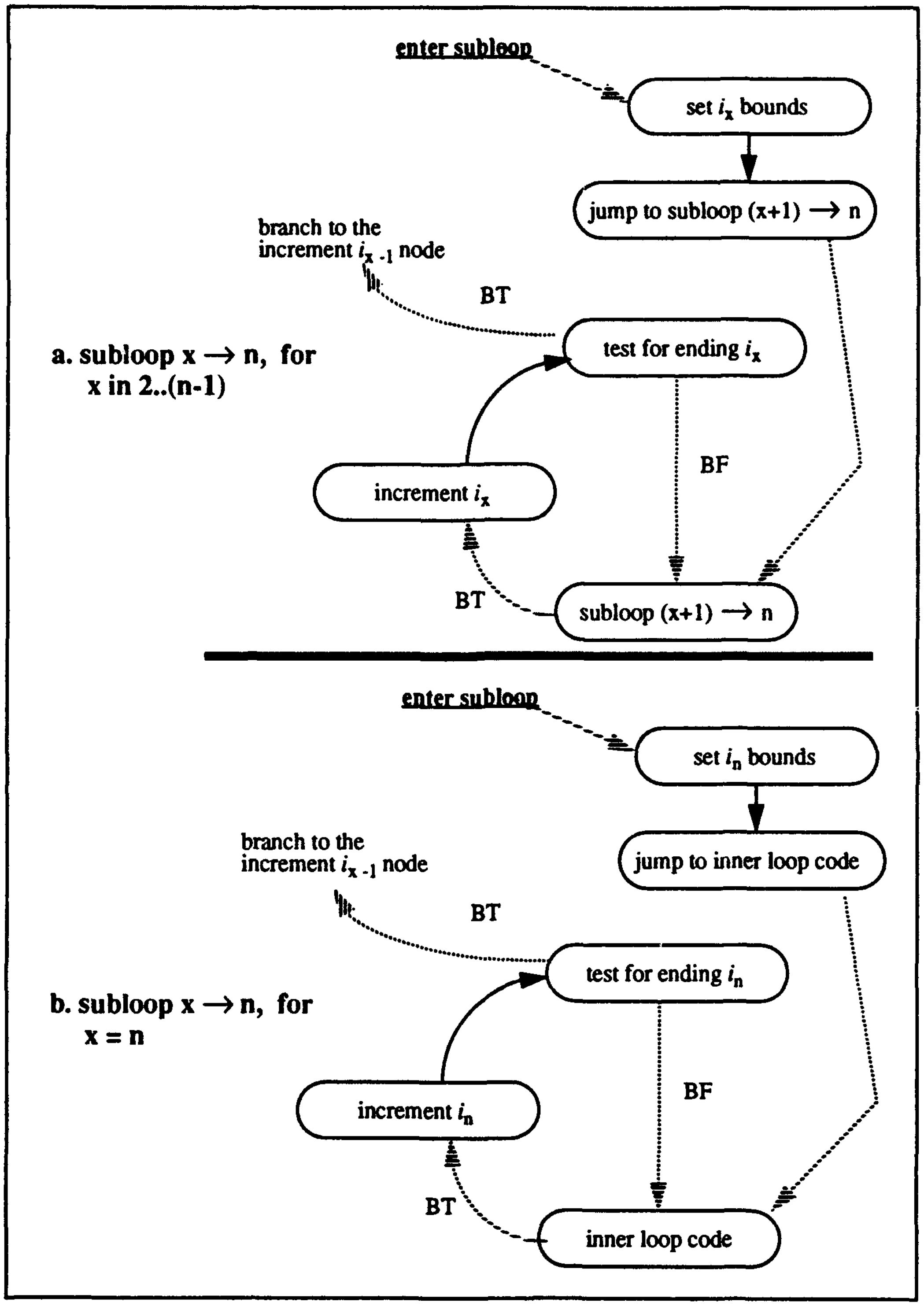


Figure 44: Recursive Definition for Subloop 2 ⟶ n

b. Modelling The Final Loop Structure Code

The final loop structure that must be created by the code generation process can be represented with a code model as for the original loop structure. As stated previously, we will assume that the code generation process will be targeting a VLIW machine with special hardware support available as described in Section IV.A.2.

The diagram which represents the final loop structure code is similar to that representing the original loop structure, but must incorporate the issued which were discussed in Section IV.B. In this case, the model is as in Figure 45, with the recursive definition for the "subloop 2 $\rightarrow$ n" node shown in Figure 46.

The significant change from the original loop model, as expected, occurs only in Figure 46.b, which represents the subloop for the innermost loop, and contains the necessary code components which are required to support the use of the pipelined code. The node "execute N_{inner} non-pipelined iterations" represents the code segments necessary to execute the non-pipelined iterations as discussed in Section IV.B.4. This code can be executed in a variety of manners, but we suggest using a relatively simple and efficient method represented by Figure 47.

c. Explanation Of The Final Loop Structure Code

To better understand the mode for the final loop structure code presented in the Figure 45 through Figure 47, as well as provide the basis for the code generation algorithm, the loop structure model nodes will be described in more detail. For each node description, we will indicate the RISC assembly code instructions required for implementing the node and any additional comments. The assembly code instructions will be used to create the code for the final loop structure. For clarity, variables such as loop control indices, known constants, etc., will be identified by their normal representation instead of by explicit values, registers or address location.

An arbitrary choice of registers identifiers are referenced as necessary support the code explanation. The register numbering will start with R1. The choice of

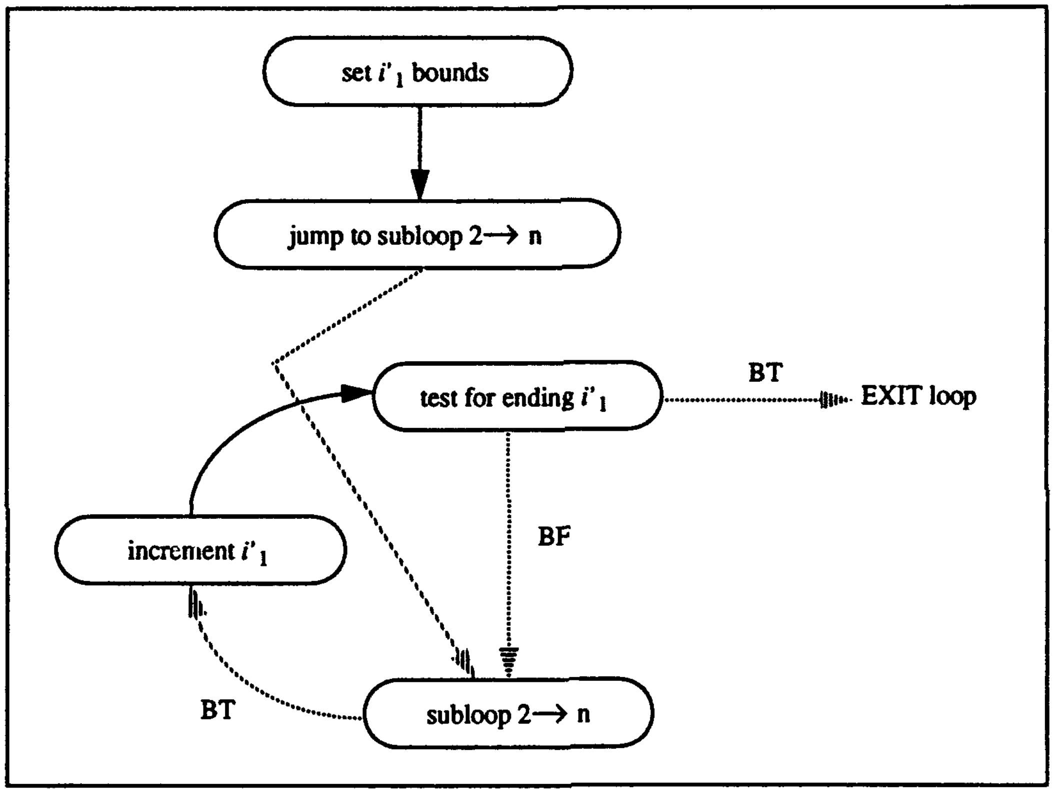


Figure 45: Final Loop Structure Code Model

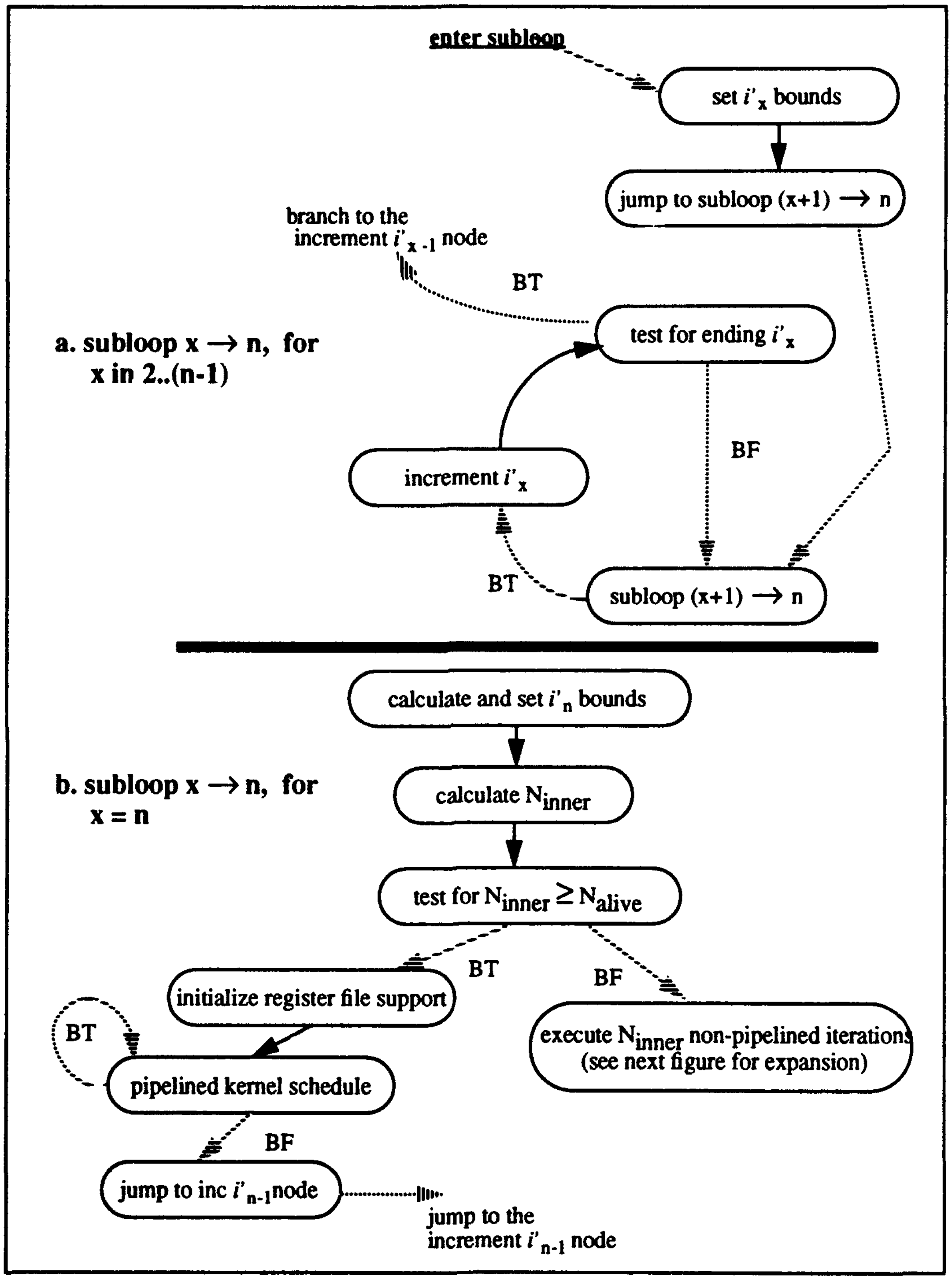
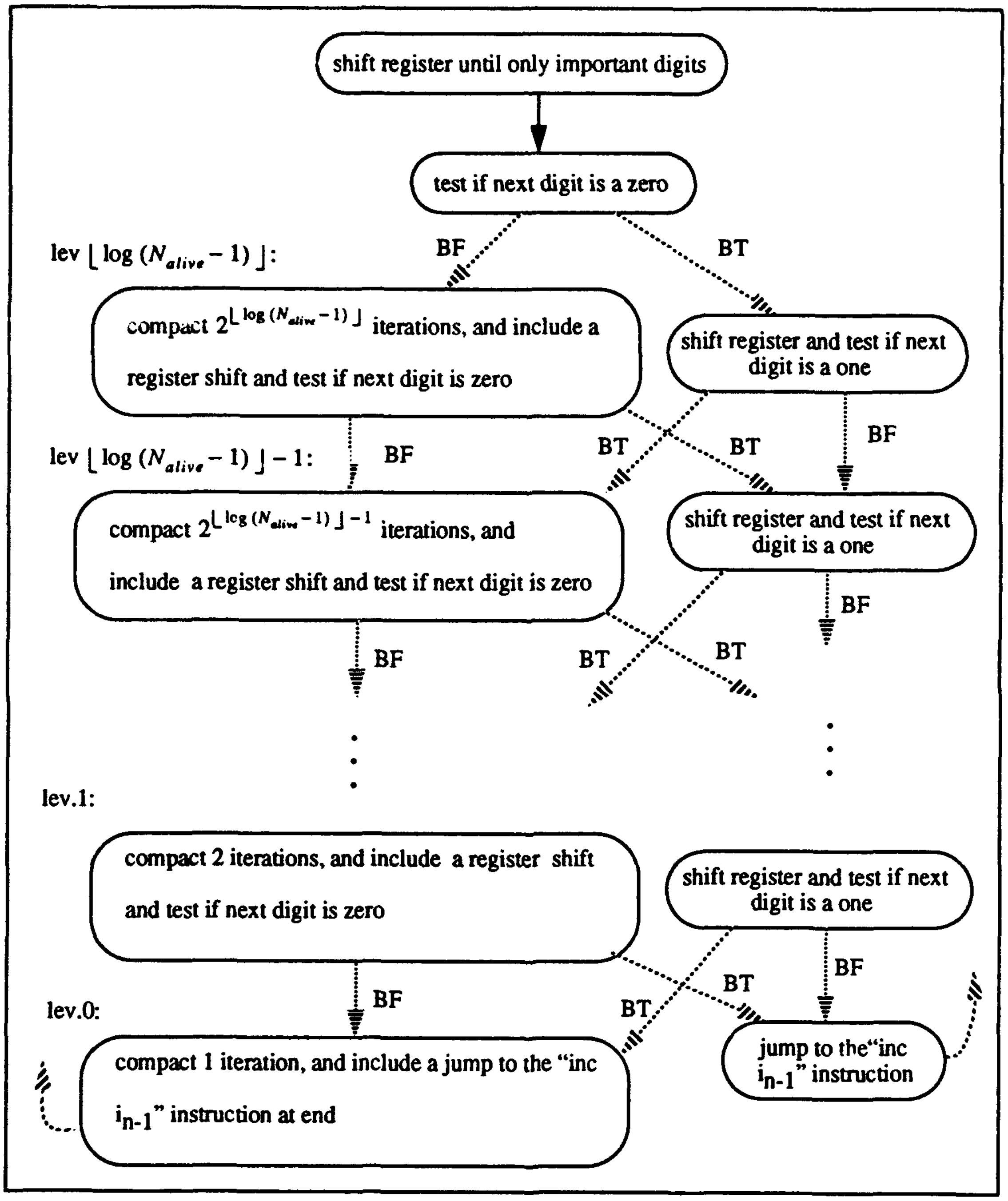

enter subloop
set i'_x bounds
jump to subloop (x+1) $\longrightarrow$ n
branch to the increment i'_{x-1} node
BT
a. subloop x $\longrightarrow$ n, for x in 2..(n-1)
test for ending i'_x
BF
increment i'_x
BT
subloop (x+1) $\longrightarrow$ n
calculate and set i'_n bounds
b. subloop x $\longrightarrow$ n, for x = n
calculate N_{inner}
test for $N_{inner} \geq N_{alive}$
BT
BF
initialize register file support
BT
execute N_{inner} non-pipelined iterations (see next figure for expansion)
pipelined kernel schedule
BF
jump to inc i'_{n-1}node
jump to the increment i'_{n-1} node

Figure 46: Recursive Definition for Subloop 2 $\longrightarrow$ n



shift register until only important digits
test if next digit is a zero
lev $\lfloor \log (N_{alive} - 1) \rfloor$:
BF
BT
compact $2^{\lfloor \log (N_{alive} - 1) \rfloor}$ iterations, and include a register shift and test if next digit is zero
shift register and test if next digit is a one
lev $\lfloor \log (N_{alive} - 1) \rfloor - 1$:
BF
BT
BT
BF
compact $2^{\lfloor \log (N_{alive} - 1) \rfloor - 1}$ iterations, and include a register shift and test if next digit is zero
shift register and test if next digit is a one
BF
BF
BT
BT
lev.1:
compact 2 iterations, and include a register shift and test if next digit is zero
shift register and test if next digit is a one
BF
BT
BF
lev.0:
BT
compact 1 iteration, and include a jump to the "inc i_{n-1}" instruction at end
jump to the "inc i_{n-1}" instruction

Figure 47: Expansion of "execute N_{inner} non-pipelined iterations" Node

registers, though arbitrary in name, ensure proper data dependences are maintained in the supporting code. As a result, although the register naming is arbitrary, the reuse of register names is significant in maintaining the correct references between variable values. Renaming of the registers used in the supporting code described in this section to order to match the available register names of a target machine is satisfactory as long as the mapping ensures the dependences are not violated (a one-to-one mapping, for example, is satisfactory).

My initial assumption that the target machine has a large number of available registers was meant to ensure that the minimal register requirements presented in the technique are supported. Not counting the registers required for loop control variables, the supporting code requires at most fourteen registers, only two of which need lifetimes which span across the loop body.

Additionally, the register naming scheme used in the code segments are done so without considering the register naming used in the original innermost loop code. As a result, some inconsistencies or register reuse problems may exist between the registers used in the supporting code and those used in the loop body. However, this problem exists only for those register variables calculated prior to the loop body and used after the execution of the loop body--that is, those registers which have a lifetime which extends beyond one innermost loop execution. This limits the problem to the registers holding the loop control variables and to that values of the variables stored in registers R8 and R10 as used below.

To alleviate the problem with loop control variables, the procedure for code generation takes as input the register assignments for the loop control variables which are used in the loop body of the modified transformed loop body. These register assignments then replace the control variable indicators in the supporting code described below. In this way, the supporting code segments and the loop body can be compatible with respect to loop control variable referencing. To avoid confusion during the explanation of the code segments, register names are not specifically assigned for loop control variables in the sample code given below. Rather, we will refer to the loop indices in the manner "i'_x".

To alleviate any problems between use of the R8 and R10 registers in the segments described below and a conflicting usage in the loop body, the procedure for generating the final code will take as input the names of two available registers to be used instead of R8 and R10 in the code segments discussed below. These register names will replace R8 and R10 in the final code and the conflict will again be avoided. The register names of "R8" and "R10" will, however, continue to be used below during the discussion.

Constant values which are determined as part of the transformation process (such as the *sf)* will be referenced using the constants name. The RISC instructions are those of the type available in the DLX type machine as explained by Hennessy and Patterson [Ref. 13], with the added instructions of BRTOP, INIT, and SET_UP as described previously for support of the special hardware. Labels which identify specific code segment will be referred to in quotes.

(1) Node "set i'_x bounds" for x = 1..(n-1) of Figure 45 and Figure 46.a. These node represents the code necessary for initializing the bounds for the any loop variable i'_x, including the i'_1 loop variable. Because we assume that the original loop structure is at least two dimensional, we are assured that the bounds of these loops are known prior to the time of execution, and therefore, the values can be considered to be the constants M'_x and N'_x, where the variable i'_x ranges from M'_x to N'_x. For all loops 1..(n-1) the values for these with respect to the original loop control bounds are known. That is:

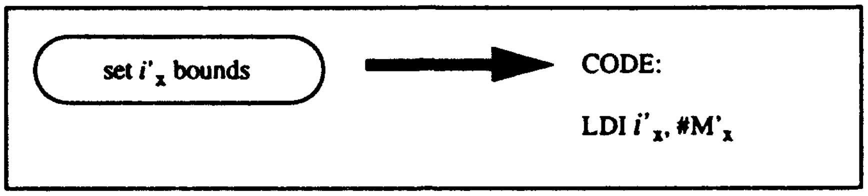
for x = 1..(n-2)

$M'_x = 1$ and $N'_x = N_x$

for x = n-1

$M'_x = (sf + 2)$ and $N'_x = [(sf+1) \times N_{n-1} + N_n]$

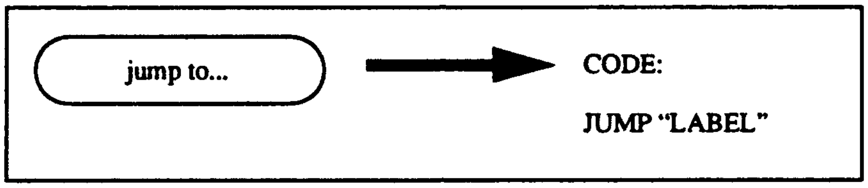
The code for this node is shown in Figure 48, with the variable E_x referring to the ending value for the variable.



```
CODE:
LDI i'x, #M'x
```

Figure 48: Explanation of the "set i'_x bounds" Nodes

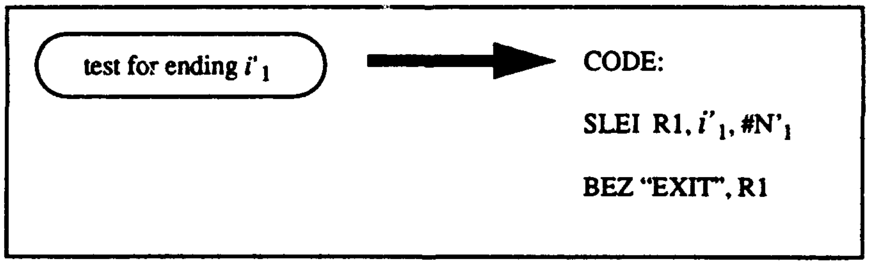
(2) Node "jump to..." of all figures. These node represents the code necessary for jumping to a new position. The jump locations will be identified by the code segment labels, so that the code is that of Figure 49.



```
CODE:
JUMP "LABEL"
```

Figure 49: Explanation of the "jump to..." Nodes

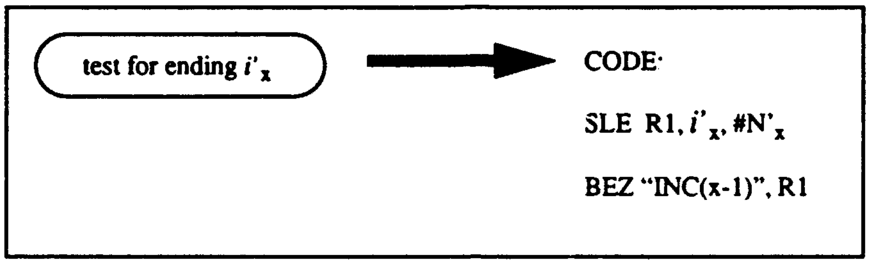
(3) Node "test for ending i'_1" of Figure 45. The test for the ending condition of i'_1 merely require to test whether the present value of the control variable has been incremented beyond the final value. As a result, Figure 50 identifies the code needed. In this case "EXIT" is the label of the label of the code which commences following the completion of the entire loop structure. R1 is the first register required by the supporting code.



```
CODE:
SLEI R1, i'1, #N'1
BEZ "EXIT", R1
```

Figure 50: Explanation of the "test for ending i'_1" Node

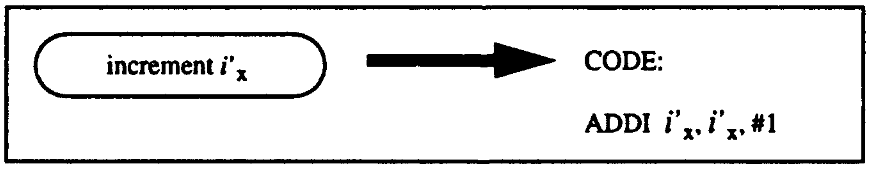
(4) Node "test for ending i'_x" for x = 2..(n-1) of Figure 46.a. As in the previous node, this node represents test whether the present value of the control variable has been incremented beyond the final value. As a result, Figure 51 identifies the code needed. In this case "INC(x-1)" represents the label which is used to branch to the code which performs the incrementation of the i'_{x-1} control variable. R1 is reused as the register containing the results of the compare operation, not expecting two comparisons to be performed at the same time.



```
CODE·
SLE R1, i'x, #N'x
BEZ "INC(x-1)", R1
```

Figure 51: Explanation of the "test for ending i'_x" Nodes

(5) Node "increment i'_x" for x = 1..(n-1) of Figure 45 andFigure 46. The increment of the control variable is simple an incrementation by one, as shown in Figure 52.

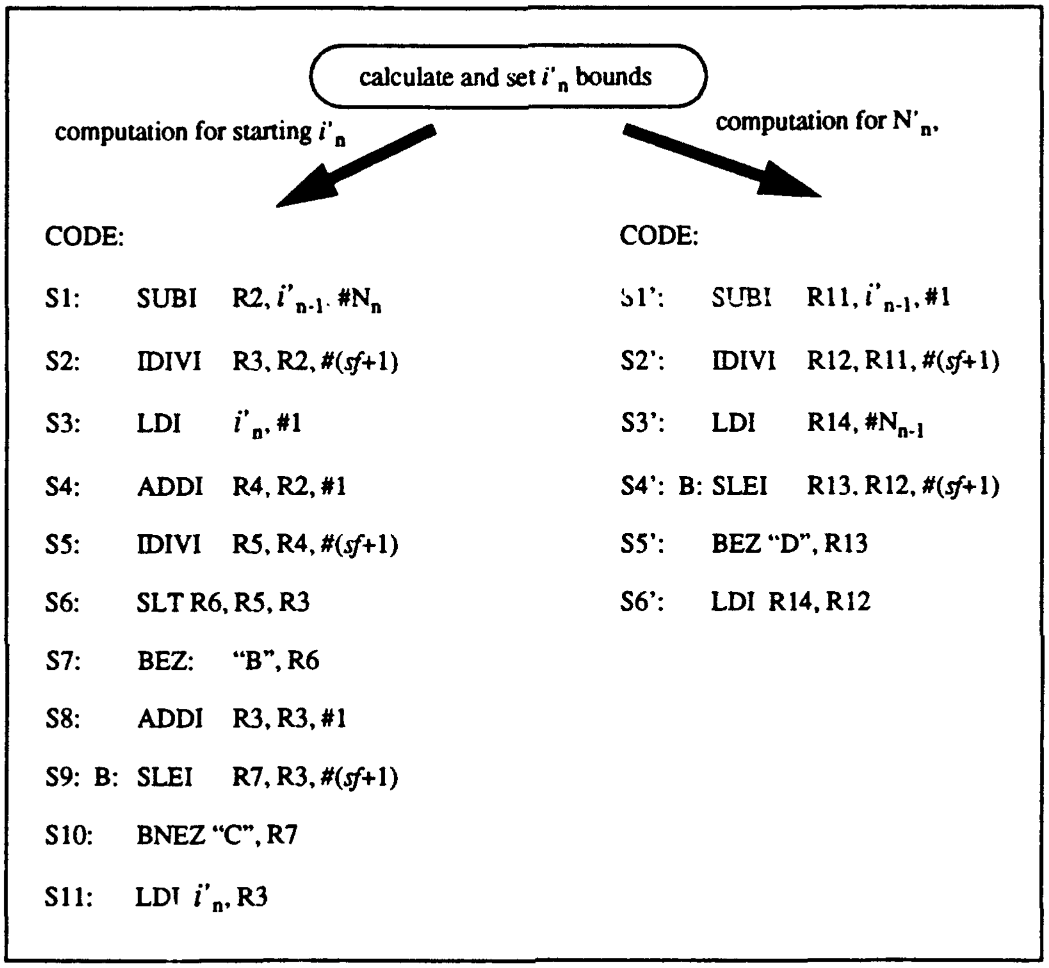


```
CODE:
ADDI i'x, i'x, #1
```

Figure 52: Explanation of the "increment i'_x" Nodes

(6) Node "calculate and set i'_n bounds" Figure 46.b. Because of the skewing process which took place during the loop structure transformation, the loop bounds of the innermost loop are dependent on the value of the i'_{n-1} variable. Hence, the calculation of bounds was discussed in Section III.A.3, with the range of i'_n being given $max(1, \left\lceil \frac{i'_{n-1} - N_n}{sf+1} \right\rceil) \ldots min(\left\lfloor \frac{i'_{n-1} - 1}{sf+1} \right\rfloor, N_{n-1})$. The floor and ceiling calculations can be performed by using an integer divide instructions, and, for the ceiling, an additional

comparison. The result is two independent calculations which determine the setting of the starting and ending i'_n values. The code which will perform this calculation is shown in Figure 53. Statement numbers are identified for each of the code segments. Labels are included ("B", "C", and "D" as needed for branches to other parts of the code.


calculate and set i'_n bounds
computation for starting i'_n
computation for N'_n.

CODE:

```
S1:     SUBI   R2, i'n-1, #Nn
S2:     IDIVI  R3, R2, #(sf+1)
S3:     LDI    i'n, #1
S4:     ADDI   R4, R2, #1
S5:     IDIVI  R5, R4, #(sf+1)
S6:     SLT R6, R5, R3
S7:     BEZ:   "B", R6
S8:     ADDI   R3, R3, #1
S9: B:  SLEI   R7, R3, #(sf+1)
S10:    BNEZ "C", R7
S11:    LDI i'n, R3
```

CODE:

```
S1':    SUBI   R11, i'n-1, #1
S2':    IDIVI  R12, R11, #(sf+1)
S3':    LDI    R14, #Nn-1
S4': B: SLEI   R13, R12, #(sf+1)
S5':    BEZ "D", R13
S6':    LDI R14, R12
```

Figure 53: Explanation for "calculate and set i'_n bounds" Node

Dependency graphs for these code segments are shown in Figure 54. The latencies for the instructions are assumed to be consistent with those of the example. The immediate loads (LDI), however, are only expected to take one time unit. When the specific capabilities of the target machine are identified, the graphs can be used to compact the code from both of the above independent computations to best utilize the resources for that node

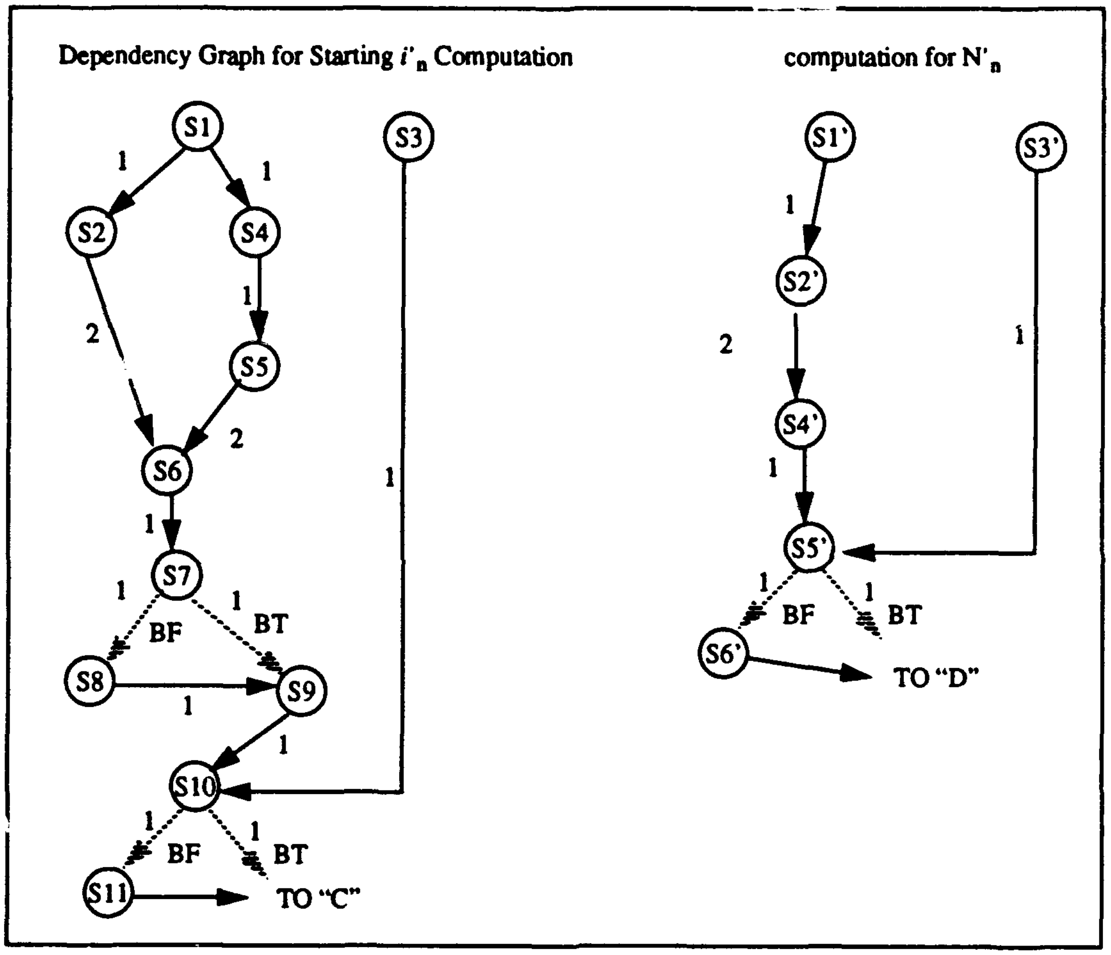
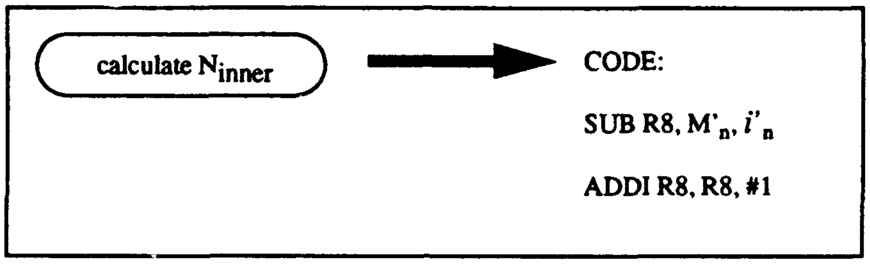


Figure 54: Dependency Graphs for i'_n Bound Calculation

(7) Node "calculate N_{inner}" of Figure 45.b. Calculating the number of innermost loop iteration is merely a matter of using the difference in the bound values. Hence, the instruction are per Figure 55.



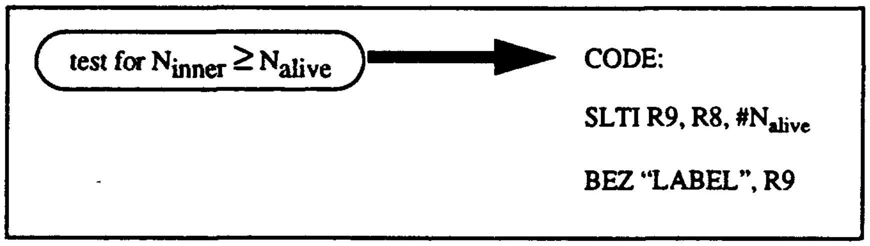
CODE:

SUB R8, M'_n, i'_n

ADDI R8, R8, #1

Figure 55: Explanation of the "calculate N_{inner}" Node

(8) Node "test for $N_{inner} \geq N_{alive}$" of Figure 45.b. This node represents the check to verify that the pipelined schedule can be used for the innermost loop. Hence, the instruction are per Figure 56. The label in the branch instruction directs the control to the segment of code executing non-pipelined iterations.



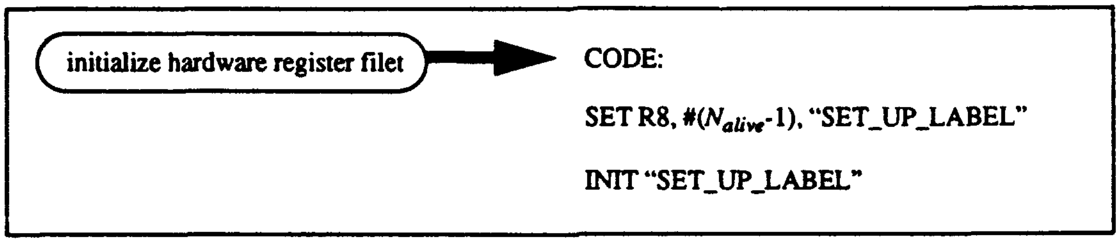
CODE:

SLTI R9, R8, #N_{alive}

BEZ "LABEL", R9

Figure 56: Explanation of the "test for $N_{inner} \geq N_{alive}$" Node

(9) Node "initialize hardware register file" of Figure 45.b. This node represents initialization instructions that must be executed as discussed in Section IV.B.2 and Section IV.B.3. The initialization consists of the setting of the LC and ESC counter values, and the triggering of the hardware register file support. The instructions for this node are per Figure 57. The label in the branch instruction refers to the label given the specifications for the register files, not a jump location.



CODE:

SET R8, #(N_{alive}-1), "SET_UP_LABEL"

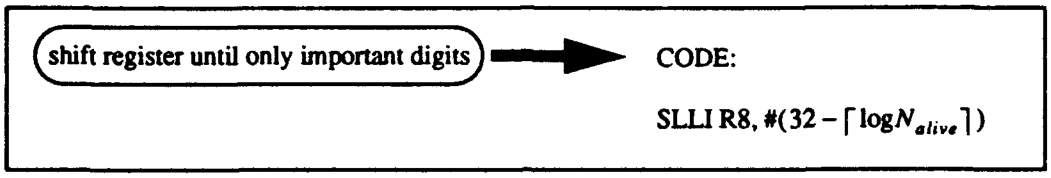
INIT "SET_UP_LABEL"

Figure 57: Explanation of the "test for $N_{inner} \geq N_{alive}$" Node

(10) Node "pipelined kernel schedule" of Figure 45.b. This node represents the code created as the pipelined kernel schedule. This is created via the separate process as discussed in Section III.B and Section IV.B. It is assumed that this code is created as part of a separate process to be used in the code generation, and will be used when putting together the final code structure, but is not discussed again here.

(11) Node "execute N_{inner} non-pipelined iterations" of Figure 45.b. This node represents the code used to execute the non-pipelined segment of code, and is further broken down in the nodes discussed for the Figure 47. The procedure represented in Figure 47 sequentially checks the important bits of the value of N_{inner} (contained in register R8) to verify if a certain power of two iterations needs to be executed. The procedure then executes a compact version of the correct number of non-pipelined iterations, and then checks the next bit for possible additional iterations.

(12) Node "shift register until only important digits" of Figure 47. This node represents the initial step of executing the non-pipelined code by shifting all of the bits of register R8 (containing the value of N_{inner}) to the left by an amount of equal to $\lceil \log N_{alive} \rceil$ (log is base two). This will leave only those bits which may have information about the value of N_{inner}, and can be calculated as a constant prior to the procedure. we assume that 32 bit words are used, so the shift must move $32 - \lceil \log N_{alive} \rceil$ bits. The resultant code is shown in Figure 58.

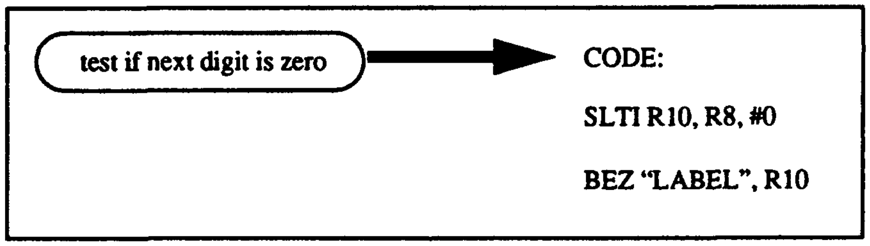


CODE:

```
SLLI R8, #(32 - ⌈logN_alive⌉)
```

Figure 58: Explanation of the "shift register until only important digits" Node

(13) Node "test if next digit is a zero" of Figure 47. This node represents the code for testing the left most digit of R8, which contains the information about how many iterations must be executed not using the pipelined schedule. The value in the register is merely checked to see if it is positive or negative. If negative, the digits is one, and it is known that at least $2^{\lfloor \log (N_{alive} - 1) \rfloor}$ iterations must be executed, and a branch is taken to that code (the label in the branch refers to that code segment). The resultant code is shown in Figure 59.

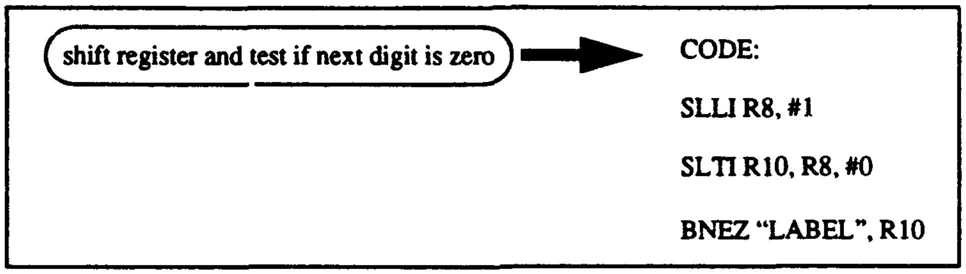


CODE:

```
SLTI R10, R8, #0
BEZ "LABEL", R10
```

Figure 59: Explanation of the "test if next digit is a zero" Node

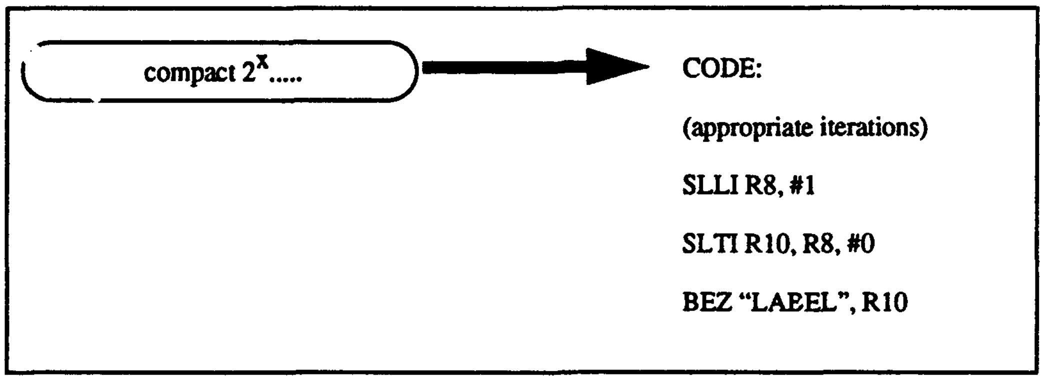
(14) Node "shift value and test if next digit is a one" of Figure 47. This node represents code executed if the previous digit of R8 that was tested was a zero. The bits in the register R8 are now shifted left one digit and the value is again tested for negative. This time, if negative, it is known that at least $2^{\lfloor \log (N_{alive} - 1) \rfloor - 1}$ additional iterations must be executed, and a branch is taken to that code (identified by the branch reference label). The resultant code is shown in Figure 60.



```
CODE:
SLLI R8, #1
SLTI R10, R8, #0
BNEZ "LABEL", R10
```

Figure 60: Explanation of the "shift register and test if next digit is a one" Nodes

(15) Node "compact 2^x iterations, and include a register shift and test if next digit is zero" where x ranges from $2..\lfloor \log(N_{alive} - 1) \rfloor$, of Figure 47. This node represents code executing a number of non-pipelined iterations. The iterations used must be those represented by the transformed loop, without the normal loop control variable increment, compare and branch. That is, they must include the transformation equations added to the loop. The additional piece of code for the register shift and value check is described in Figure 61. The label for the branch identifies the piece of code for the which is executed if the resultant value in R8 is positive, sending the control back to a testing code segment as explained in Section IV.C.1.c.14. The code shown is not compacted, but compaction of the code would result in greater efficiency.



```
CODE:
(appropriate iterations)
SLLI R8, #1
SLTI R10, R8, #0
BEZ "LABEL", R10
```

Figure 61: Explanation of the "compact 2^x iterations, and include a register shift and test if next digit is zero" Nodes

(16) Node "compact 1 iteration, and include a jump to the "inc i'_{n-1}" instruction" of Figure 47. This node represents code executing one non-pipelined iterations, compacted and includes a jump back to the "inc i'_{n-1}" instruction. The instructions for the node are shown in Figure 62. The "LABEL" of the jump indicates the label for the "inc i'_{n-1}" instruction.

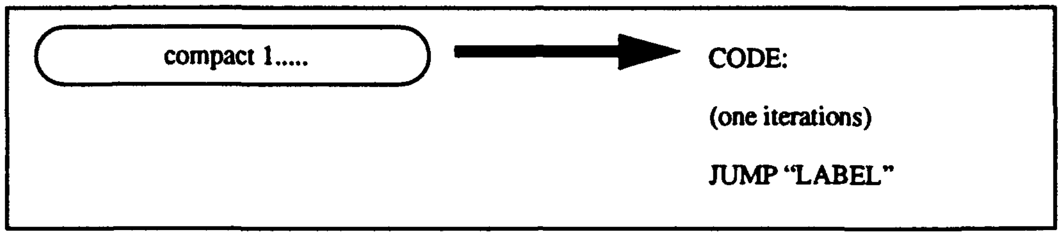


Figure 62: Explanation of the "compact 1 iterations, and include a jump to the "inc i'_{n-1}" Node

2. The Final Code Generation Process

Using the model of the final loop code structure and incorporating the issues of code generation brought up in Section IV.B, a code generation process has been created for manufacturing the final loop code structure which uses the loop pipelining technique presented in this thesis.

The sections below list the required initial conditions an the process for code generation.

a. Initial Conditions For Code Generation Process

The initial conditions, assumptions, and support for performing the code generation are as follows:

- It is assumed that the word size is 32 bits in the calculations for register shifting amounts
- The dimension of the original loop structure is known, designated as "n"
- The values of the original loop structure control variable bounds are known, and are contained in the array N[x] where x ranges from 1..n (arrav being N[1]..N[n]).

- To allow flexibility to the desired reference syntax to the index variables, the correct labels for the indices will be the values assigned to the array I[x], with x ranging from 1..n, so that the reference symbol for i'_1 will be the value contained in the element I[1].
- A label which specifies the requirements for the set up for the register files is identified and will be used to pass into the procedure for referencing when the requirements are to take effect.
- Two registers that are free to be used without interfering with the pipelined code are identified to replace the R8 and R10 registers in the supporting code. The register identifiers are passed in as values to the parameters Y and Z, with default values of R8 and R10 respectively.
- a function is made available to compact the computation for the inner loop bounds. The function will be referred to as COMPACT_COMPUTATIONS, and uses the graphs as specified in Figure 54 and the resources specified to generate compacted code. The function takes as arguments the register labels contained in I[n-1] and I[n], as well as the values of N[n], and *sf*. It returns the compacted code segment for insertion into the final code.
- a function is made available to compact a specific number of iterations. The function is called MULTIPLE_COMPACTION and takes as input arguments the final loop DDG for a single iteration as in Figure 30, the number of iterations that need to be included in the compaction, and the branch destination label following a true result of the testing of the R8 register value. The function should eliminate the branches the individual iterations, connect the individual iterations via the loop variable increment instructions. The compaction should also include the necessary register shift on R8 and test for next action, ending with the branch to the correct code segment location. Register renaming for the sequential segments of code is also necessary to allow some overlap of register usage between iterations. Returned is that block of compacted code which can then be inserted into the generated code.
- a function is made available to compact a single iteration. The function is called SINGLE_COMPACTION and takes as input arguments the transformed modified dependency graph (as in Figure 19) for a single iteration and the code segment label to be jumped to after the code block completed. The function should

eliminate the branches the individual iteration, compact the iteration, and insert the necessary jump to the outer loop control as the last instruction of the code block generated.

b. The Final Code Generation Process

The final code generation process given below includes in its description the application of the wavefront transformation, as well as the application of the modulo scheduling procedure. In this way, the code generation process incorporates the used of the loop pipelining technique presented in this thesis as the preliminary steps required to create the pipelined kernel schedule, provide needed values of *sf* and N_{alive} for use in the coding generation algorithm, and provide the modified transformed DDG for use with the iteration compaction procedures.

Application of the code generation algorithm is the last step in the code generation process, and is used to write (to some destination) the revised RISC assembly type code which has been modified to include the appropriate code segments needed to support the transformation and pipelined schedule. Because the output is expected to be used for a VLIW machine, those sub-instructions which can be executed in the same VLIW instruction should be written on the same line, or use some other method of indicating assignment to specific VLIW instructions. The code generation algorithm is given in pseudo code format. The procedure "write" specifies the sub-instructions that needs to be written to the current VLIW instruction, and assigns it to the correct available resource. If dependencies do not prohibit instructions from being included in the same VLIW instruction, then consecutive "write" commands are issued. Sub-instructions groups which are dependent are separated by a "new_line" command, to explicitly indicate that dependencies require that the instruction must belong to the next VLIW instruction. If the argument for "write" procedure is in double quotes, then the included text should be written verbatim. If the argument is not in quotes, then the text identifies a variable whose value should be written. The ampersand symbol ("&") is used for concatenation of objects. For example, if the write statement is: write("R3, R4, #" & X) where X=3, then the written output should be ADDI R3, R4, #3.

The command "write_label" is used to indicate a code segment label assignment for the subsequent code, and is merely written as the identifier, not as code.

(1) The Code Generation Process. The code generation process is summarized as follows:

- Apply the Wavefront Transformation Procedure to create the modified transformed DDG, with loop variable incrementation instructions added
- Apply The Acyclic DDG Modulo Scheduling Technique
- Create The Pipelined Kernel Schedule
- Apply the GENERATE_CODE algorithm as shown is section (2) below.

(2) The Code Generation Algorithm. The code generation algorithm is named GENERATE_CODE and is given as follows:

```
algorithm GENERATE_CODE (    input: n, sf, N_alive, array N[X], array I[X], target
                                    machine resources, set-up label
                                    for hardware specification for
                                    register files, register identifier to be
                                    used as R8 with default as R8 (ref-
                                    erence as variable Y), register
                                    identifier to be used as R10 with
                                    default as R10 (reference as var-
                                    iable Z);
                             output: final code)
begin
    --set first control variable bounds
    if n>2 then
        write("LDI" & I[1] & ", #1")
        write("JUMP LOOP2")  --note: this instruction can be combined with the
                                   --above instruction if resources allow
        new_line
        write_label("INC1:")
        write("ADDI" & I[1] & "," & I[1] & ", #1")
        new_line
        write_label("TEST1:")
        write("SLEI R1," & I[1] & ", #" & N[1])
        new_line
        write("BEZ EXIT, R1")
        new_line
    else --n=2 so that bounds must be adjusted
        START := sf + 2
        END := (sf+1)*N[1]+N[n]
        new_line
        write("LDI" & I[1] & ", #" & START)
        write("JUMP LOOP2)
```

```
     new_line
     write_label("INC1:")
     write("ADDI" & I[1] & "," & I[1] & ", #1")
     new_line
     write_label("TEST1:")
     write("SLEI R1," & I[1] & ", #" & END)
     new_line
     write("BEZ EXIT, R1")
     new_line
end if

--set rest of loop control structure for outer n-2 iterations
for X in 2..(n-2) loop  --only will execute if n>=4
     write_label("LOOP" & X & ":")
     write("LDI"& I[X] & ", #1")
     write("JUMP LOOP" & (X+1))
     new_line
     write_label("INC" & X & ":")
     write("ADDI" & I[X] & "," & I[X] & ", #1")
     new_line
     write_label("TEST" & X & ":")
     write("SLEI R1," & I[X] & ", #" & N[X])
     new_line
     write("BEZ INC" & (X-1) & ", R1")
     new_line

if n>2 then --control for loop second innermost loop not yet done
     START := $sf$ + 2
     END := ($sf$+1)*N[1]+N[n]
     write_label("LOOP" & (n-1) & ":")
     write("LDI" & I[n-1] & ", #" & START)
     write("JUMP LOOP" & n)
     new_line
     write_label("INC" & (n-1) & ":")
     write("ADDI" & I[n-1] & "," & I[n-1] & ", #1")
     new_line
     write_label("TEST" & (n-1) & ":")
     write("SLEI R1," & I[n-1] & ", #" & END)
     new_line
     write("BEZ INC" & (n-2) & ", R1")
     new_line
end if

--code of innermost loop
     write_label("LOOP" & n & ":")
     --compact the boundary calculation code with called procedure
     CODE_SEGMENT = COMPACT_CODE(boundary code graphs, available
                                              resources)
     write(CODE_SEGMENT)
     new_line
     write_label("D")
     --determine the number of inner loop iterations
     write("SUB " & Y ", " & R14 & "," & I[n])
     new_line
     write("ADDI" & Y & "," & Y & ", #1")
     new_line
     --determine if the pipelined kernel can be used
     write("SGEI R9, " & Y & ", #" & $N_{alive}$)
     new_line
     write("BEZ TRANS, R9")
     new_line
```

```
--initialize the hardware register files and counters
write("SET " & Y & ", #" & ($N_{alive}$-1) & "," & SET_UP_LABEL)
new_line
write("INIT" SET_UP_LABEL)
--insert the pipelined kernel schedule
write_label("LOOP_TOP:")
write(PIPELINED_KERNEL_SCHEDULE)
new_line
write("JUMP INC" & (n-1))
new_line

--execute the non-pipelined code segments
    --calculated needed values
    FIRST_SHIFT := 32 - CEILING[log($N_{alive}$)]
    MAX_LEVEL := FLOOR[log($N_{alive}$-1)]
    --shift the register Y until important bits
    write_label("TRANS:")
    write("SLLI " & Y ", #" & FIRST_SHIFT)
    new_line
    --test the next bit
    write("SLTI" & Z & ", " & Y & ", #0")
    new_line
    write("BEZ SHIFT" & MAX_LEVEL & ", " & Z)
    new_line

    --compacted iterations
    for X in 1..MAX_LEVEL reverse loop
        write_label("LEV" & X & ":")
        write(MULTIPLE_COMPACTION(dependency graph, $2^X$,
          "SHIFT" & (X-1)))
    new_line
    end loop
    write_label("LEV" & 0 & ":")
    write(SINGLE_COMPACTION(dependency graph, "INC" & (n-1)))
    new_line

    --shift and tests
    for X in 1..MAX_LEVEL reverse loop
        write_label("SHIFT" & X & ":")
        write("SLLI" & Y ", #1")
        new_line
        --test the next bit
        write("SLTI" & Z & Y ", #0")
        new_line
        write("BNEZ LEV" & (X-1) & ", " & Z)
        new_line
    end loop
    write_label("SHIFT0:")
    write("JUMP INC" & (n-1))
    new_line
    write_label("EXIT:")
```

(3) An Example Of Resultant Code Produced. As an example of the expected output code from the GENERATE_CODE algorithm, assume that n=5, sf=2, N_{alive}=3, with all N[x]= 100 and all I[x] = i_x. Additionally, assume that both R8 and R10 are

used in the loop body, but R24 and R25 are free, so Y := R24 and Z := R25. Assume also that there are two fully capable processors, and the INIT and SET commands for initializing the register files are capable of being executed on any unit. Then resultant output from the above algorithm appears as follows, with code generated from compaction or modulo scheduling procedures bolded and italicized, and each text line indicating the VLIW sub-instructions that can be executed:

```
    LDI i1, #1                        JUMP LOOP2
INC1:
    ADDI i1, i1, #1
TEST1:
    SLEI R1, i1, #100
    BEZ EXIT, R1
LOOP2:
    LDI i2, #1                        JUMP LOOP3
INC2:
    ADDI i2, i2, #1
TEST2:
    SLEI R1, i2, #100
    BEZ INC1, R1
LOOP3:
    LDI i3, #1                        JUMP LOOP4
INC3:
    ADDI i3, i3, #1
TEST3:
    SLEI R1, i3, #100
    BEZ INC2, R1
LOOP4:
    LDI i4, #4                        JUMP LOOP5
INC4:
    ADDI i4, i4, #1
TEST4:
    SLEI R1, i4, #400
    BEZ INC3, R1
LOOP5:
    CODE_SEGMENT FROM COMPACT COMPUTATIONS
D:
    SUB R24, R14, i5
    ADDI R24, R24, #1
    SGEI R9, R24, #3
    BEZ TRANS, R9
    SET R24, #2, SET_UP_LABEL)
    INIT SET_UP_LABEL
LOOP_TOP:
    PIPELINED_KERNEL_SCHEDULE
    JUMP INC4
TRANS:
    SLLI R24, #30
    SLTI R25, R24, #0
    BEZ SHIFT1, R25
LEV1:
    MULTIPLE_COMPACTION(dependency graph, 2, SHIFT0)
LEV0:
    SINGLE_COMPACTION(dependency graph, INC4)
SHIFT1:
        SLLI R24, #1
        SLTI R25, R24, #0
        BEZ LEV0, R25
    SHIFT0:
        JUMP INC4
    EXIT:
```

3. An Example Application Of The Code Generation Process

The example which has been used throughout this thesis is used once again to complete the explanation of the code generation process. For the example, n=2, $sf=1$, $N_{alive}=3$, and N[x]= {100, 500}. To be consistent with the original code presented in Section IV.B.1., the index variables are passed in as I[X] = {R15, R1[0]}, where R1[0] is the first element of the register file R1[X] created for the renaming of the R1 variable. In this way, the R1[0] register is used for all code outside of the pipelined loop, therefore being automatically initiated to the correct value when the pipelined kernel starts.

Assume that the information about the register file requirements were established under the label SETTINGS, where the requirements for the renamed registers and the ICR are created.

Lastly, the register R8 and R10 are used in the loop body, hence R20 and R21 will be passed in as free registers to be used in place of the variables Y and Z.

We will assume again that the INIT and SET instructions can be performed by any functional unit available.

a. *Compacting Non-Pipelined Iterations*

Before presenting the final loop code structure for the example, the compacted non-pipelined iterations which would result from the presumed to exist functions COMPACT_COMPUTATIONS, MULTIPLE_COMPACTION, and SINGLE_COMPACTION must be determined. These functions can be based on any sequential code compaction process, such as those discussed by Colwell, Nix, O'Donnel, Papworth and Rodman [Ref. 14]. For the purposes of the example, we have generated the resultant code segments which might have resulted from a compaction process.

(1) For COMPACT_COMPUTATIONS. For the compaction of the boundary computation of the innermost loop boundaries, Figure 63 shows one compaction schedule which is adequate.

		Resource Unit				
time	label	adder	adder	multiplier	Load/Store	Branch
1		S1	S1'		S3	
2		S4		S2	S3'	
3				S5		
4				S2'		
5		S6				
6		S4'				S7
7	B:	S8				
8		S9				
9						S10
10					S11	
11	C:					S5'
12					S6'	

Figure 63: Example Code Compaction for Innermost Loop Bounds Computation Segment

(2) For SINGLE_COMPACTION. The result of this procedure is a single piece of compacted code. Using the final modified transformed DDG from Figure 19, an adequate compacted segment of code is given in Figure 64, with the JUMP to INC1 labelled code segment included.

		Resource Unit				
time	label	adder	adder	multiplier	Load/Store	Branch
1		S5		S13		
2		S15		S1		
3		S14		S6		
4		S2	S7			
5		S3	S8			
6		S11			S4	
7					S9	
8						
9		S10				
10					S12	JUMP INC1

Figure 64: Compacted Single Iteration

(3) For MULTIPLE_COMPACTION. The result of this procedure is a single piece of compacted code for multiple iterations. The pieces of code generated are those for groups of iterations from 2..$\lfloor \log (N_{alive} - 1) \rfloor$ (log in base two). However, the value of $\lfloor \log (N_{alive} - 1) \rfloor$ is two. Hence, only one compacted segments for multiple iterations will be necessary, containing two iterations. This compacted piece of code is shown in Figure 65.

The different iterations are indicated by subscript of "1" or "2". The additional instructions for the identifying the next code segment are included as the SLLI, SLTI, and BEZ instructions. The R20 and R21 registers are substituted for the R8 and R10 as designated in the procedure input. Not covered or shown is the need of some register renaming of the compacted code as appropriate to ensure no interference of register reusage between the iterations themselves.

		Resource Unit				
time	label	adder	adder	multiplier	Load/Store	Branch
1		$(S15)_1$	$(S5)_1$	$(S13)_1$		
2		$(S5)_2$	$(S14)_1$	$(S13)_2$		
3		$(S14)_2$	$(S2)_1$	$(S1)_1$		
4		$(S7)_1$	$(S2)_2$	$(S6)_1$		
5		$(S7)_2$	$(S3)_1$	$(S1)_2$		
6		$(S8)_1$	SLLI R20, #1	$(S6)_2$	$(S4)_1$	
7		$(S3)_2$	SLTI R21, R20, #0		$(S9)_1$	
8		$(S10)_1$	$(S11)_1$		$(S4)_2$	
9		$(S15)_2$	$(S11)_2$		$(S9)_2$	
10		$(S10)_2$				
					$(S12)_1$	
					$(S12)_2$	BEZ SHIFT0, R21

Figure 65: Compacted Code for Two Iterations

b. Creating The Pipelined Kernel Schedule

The pipelined kernel schedule which will be generated for the final code structure is that same schedule shown in Figure 31.

c. The Final Loop Code Structure

The final loop code structure produce from the process is given in the schedule shown in Figure 66. The larger sections of code that are produced through compaction or pipelining have been left out and merely referenced to help in clarity of illustration.

		Resource Unit				
time	label	adder	adder	multiplier	Load/Store	Branch
1					LDI R15, #3	JUMP LOOP2
2	INC1:	ADDI R15, R15, #1				
3	TEST1:	SLEI R1, R15, #700				
4						BEZ EXIT, R1
5	LOOP2:	INSERT COMPACTED COMPUTATION CODE SCHEDULE OF FIGURE 63				
. . .						
16						
17	D:	SUB R20, R14, R15				
18		ADDI R20, R20, #1				
19		SGEI R9, R20, #2				
20						BEZ TRANS, R9
21		SET R20, #1, SETTINGS				
22		INIT SETTINGS				
23	LOOP_TOP:	INSERT PIPELINED KERNEL SCHEDULE OF FIGURE 39				
. . .						
27						
28						JUMP INC1
29	TRANS:	SLLI R20, #31				
30		SLTI R21, R20, #0				
31						BEZ SHIFT1, R21
32	LEV1:	INSERT COMPACTED TWO ITERATIONS CODE SCHEDULE OF FIGURE 64				
. . .						
45	LEV0:	INSERT COMPACTED SINGLE ITERATIONS CODE SCHEDULE OF FIGURE 65				
. . .						
54						
55	SHIFT1:	SLLI R20, #31				
56		SLTI R21, R20, #0				
						BNEZ LEV0, R21
58	SHIFT0:					JUMP INC1
59	EXIT:					

Figure 66: Final Restructured Code Loop For Example

V. EVALUATION AND ANALYSIS

Having presented the proposed loop pipelining technique, there is an obvious need to evaluate the effectiveness of the technique as well as analyze the complexity of the code generation procedure for creating the final product. This chapter attempts to do both. An evaluation of the performance gained by the technique will be discussed first, followed by an analysis of the procedure for code generation given in the last chapter,

A. EVALUATION OF TECHNIQUE PERFORMANCE

The ideal solution to the loop pipelining problem for perfectly nested loops would be a solution which does not require the addition of modifying instructions, uses the IIII which is equal to the lower bound IIII based only on resources available (that is, it is not constrained by a greater lower bound from cyclic dependences), and does not require iteration transitioning areas, prologs, epilogs or any preconditioning iterations. In this way, the maximum utilization of the resources is obtained with no additional overhead.

The use of special hardware allows the elimination of precondition code, as noted before. The execution of prologs and epilogs, however, cannot be arbitrarily eliminated, because they are the result of overlapping iterations and are non-essential only in that unlikely case when the value for N_{alive} in a pipelined schedule is one. The existence of the prolog and epilog will in all cases reduce the utilization of the resources below that of the ideal case (because only partial pipelined schedules are being executed). As a result, the ideal case cannot generally be met, but it provides an upper bound on performance obtainable by loop pipelining. It is therefore useful as a guide in evaluating the effectiveness of a technique.

As first mentioned, the IIII which is used in modulo scheduling is a measure of the utilization of resources and the iteration throughput of the program. The lower the IIII, the greater throughput and performance. As a result, the upper bound on performance is directly related to the lower bound on the IIII. If the IIII is derived only from resource requirements, the upper bound on performance increases (i.e., it is *scalable*) with the addition of resources just as the IIII is scalable with the addition of resources. The upper bound on performance has the potential to increase with each resource addition, until enough resources are provided to specifically assigned one to each instruction. At this point, the lower bound IIII based on resources will have the value of one, and the addition of more resources will not result in increased benefit.

The scalable nature of performance with the addition of resource is made possible only if the IIII can, in fact, be lowered with the addition of more resources. Because the loop pipelining technique proposed in this thesis uses an Acyclic DDG Modulo Scheduling method, the calculation of the IIII in the technique is dependent only the available resources, just as in the ideal case. As a result, **the performance benefit of the proposed technique is also scalable with the number of resources.**

The previously proposed Modulo Scheduling techniques which could be applied to perfectly-nested loops approach the problem with a direct application of Modulo Scheduling to Cyclic DDG's (such as the techniques of Aiken and Nicolau [Ref. 5], Lam [Ref. 2], Rau, Schlansker, and Tirumalai[Ref. 6], and Zaky [Ref. 7]). Although they avoid the addition of transformation instructions and having to overcome the implications of a skewed iteration space, the resultant IIII is restricted not only by the resources and instruction types, but also by the length of the dependence cycles. Consequently, the IIII is prevented from being reduced below the limit required by the most limiting cyclic dependence, no matter what the resource availability. Use of a cyclic DDG modulo scheduling technique, therefore, eliminates the scalable performance benefit that additional resources should provide.

Although the loop pipelining technique presented in this thesis requires the addition of two instructions at the beginning of the loop body to support the wavefront transformation (as per Section III.A.3 and Section III.A.4), these instructions do not limit the performance obtainable. In the worst case, these instructions can be absorbed with no loss in performance by the addition of resources. In the best case, time which otherwise would have left resources idle can be used to execute these instructions. The real difference between the performance achievable with the pipelining method presented and the ideal case is the overhead required to compute the bounds on the innermost loop control variable and execute non-pipelined iterations.

However, we believe that in most cases the scalable nature of the performance provided by the technique presented creates an advantage over previous cyclic DDG modulo scheduling methods which exceeds the performance detriment that the overhead creates. To better understand this advantage, the example is again used to demonstrate this point.

1. The Ideal Solution For The Example

Once again, consider the example introduced in Figure 17, with the innermost loop body comprised of the original twelve instructions, and assuming two necessary instructions in each innermost iteration for loop control (as was needed when special hardware control was provided), for a total of fourteen instruction in the innermost loop code (eight Add/Sub, two multiply, three load/store, and one branch). With two adders, one multiplier, a branch unit, and a load/store unit, the upper bound the performance for loop pipelining, based purely on resource requirements, $IIII_{Lower\ Bound} = \left\lceil \frac{8}{2}, \frac{2}{1}, \frac{3}{1}, \frac{1}{1} \right\rceil = 4$.

For the ideal case, the code structure can be modelled in a similar fashion as was used in Section IV.C, with a final schedule given as in Figure 67. The registers R1 and R2 hold the value of the control variables for i_1 and i_2, respectfully, to be consistent with the instructions in Figure 17. R15 is an available register for use in the comparison at time three.

Although in reality a prolog and epilog are essential, we assume that they will not exist in the ideal case in order to achieve an maximum performance estimate for this the example.

As will be seen, by adding resources, the time units required for the IIII will decrease, and lower the overall time required for the execution of the loop.

		Resource Unit				
time	label	adder	adder	multiplier	Load/Store	Branch
1					LD R1, #1	JUMP LOOP2
2	INC1:	ADDI R1, R1, #1				
3	TEST1:	SLEI R15, R1, #100				
4						BEZ EXIT, R15
5					LDI R2, #1	
6	LOOP 2:	SET 500, #0, SETTINGS				
7		INIT SETTINGS				
8	LOOP_TOP:	PIPELINED SCHEDULE WITH IIII OF FOUR AND BRANCING BACK TO LOOP_TOP				
⋮						
11						
12	:					JUMP INC1
	EXIT:					

Figure 67: Ideal Schedule For Example Loop

Hence, the performance upper bound for this resource configuration establishes a minimum of 200,698 $(1 + \{99[(500 \times 4) + 7] + (500 \times 4 + 4)\})$ time units to execute all 50,000 inner loop iterations, yielding a lower bound on the average of 4.01 time units per iteration.

2. A Cyclic DDG Modulo Scheduling Method

If a cyclic DGG modulo scheduling technique was applied directly to the innermost loop, no additional innermost loop instructions would need to be added to the original DDG because no transformation would be required. Assuming again that special hardware support would be available, two instructions for incrementation of the incrementation of the innermost control variable would produce a total of fourteen instructions which would be pipelined as the innermost loop. The III based on resources specified would still be calculated to be four time units, **but the IIII required due to the cyclic dependences would be five time units.** This is due to the S4 to S10 to S12 cyclic dependence which requires five time units to complete between iterations (descriptions of such calculations can be reviewed in work by Zaky [Ref. 7]). As a result, the minimal IIII which must be considered is five time units. Cyclic modulo scheduling techniques as presented by Lam [Ref. 2] and Rau, Schlansker, and Tirumalai [Ref. 6] would require an iterative process to attempt to schedule the DDG into a modulo resource reservation table with five time units (in the case of Lam's technique [Ref. 2], the technique simplifies application for non-fully connected DDG's so that an IIII of five time units would even be impossible for the example). Consideration for epilog and prolog execution and register renaming are also required.

The example in Figure 17 provides a very good situation for cyclic DDG Modulo Scheduling with the resources as specified, a schedule can actually be generated in which N_{alive} equal to two, requiring only one iteration to be executed in the prolog (meaning the prolog and epilog are also limited to five time units). Renaming of one register is required, but with hardware support assumed, this does not affect the result. Once again, for cyclic

DDG modulo scheduling the code structure can be modelled in a similar fashion as was used in Section IV.C, with a final schedule given as in Figure 68.

		Resource Unit				
time	label	adder	adder	multiplier	Load/Store	Branch
1					LDI R1, #1	JUMP LOOP2
2	INC1:	ADDI R1, R1, #1				
3	TEST1:	SLEI R15, R1, #100				
4						BEZ EXIT, R15
5					LDI R2, #1	
6	LOOP 2:	SET 500, #1, SETTINGS				
7		INIT SETTINGS				
8	LOOP_TOP:	PIPELINED SCHEDULE WITH IIII OF FIVE AND BRANCING BACK TO LOOP_TOP				
⋮						
12						
13	:					JUMP INC1
	EXIT:					

Figure 68: Cyclic DDG Modulo Scheduling Final Code For Example

Hence, the performance upper bound establishes a minimum of 250,599 $(1 + \{99[(500 \times 5) + 7] + (500 \times 5 + 4)\})$ time units to execute all 50,000 inner loop iterations, yielding an average of 5.01 time units per iteration.

Because the IIII is limited by the cyclic dependences, adding resources will not alter the pipelined kernel size. As a result, the average time units per iteration cannot be lowered.

3. The Proposed Acyclic DDG Modulo Scheduling Method

By modifying the loop structure to allow the application of an acyclic DDG modulo scheduling method, the technique presented in this paper attempts to eliminate the effect of cyclic dependences and thereby remove any limit on the lower bound of IIII which the cyclic dependences would cause. The overhead required to implement the transformation are the two transformation instructions added to the innermost loop body, the loop bound calculations, and the execution of non-pipelined iterations. The transformation instructions added to the innermost loop body must be included in the schedule creation process, and may cause the lower bound to the IIII to increase, which can be countered by an increase in resources.The added loop bound calculations and the execution of the non-pipelined iterations become less significant as the number of iterations using the pipelined kernel increases.

Additional consideration for execution of a prolog and epilog are required as they are for a cyclic DDG modulo scheduling. The most significant differences in the resultant code of this technique and the others are the twofold: one, loop bound calculations are required to determine starting and stopping condition for the innermost loop bounds, and two, non-pipelined sections of code exist at the boundaries of the second innermost loop.

In the general case, the size of the code block needed to calculate the innermost loop bounds and the compacted non-pipelined blocks of code is very much machine dependent. In addition, the number of pieces of compacted iterations relies on the original loop bounds and the number of alive iterations in the final pipelined loop schedule. In the case of the example, the final loop code was shown in Figure 66.

For the example, the total number of time units required for execution is 272,314. This yields an average of 5.45 time units per iteration. A number higher than that resulting from a cyclic DDG modulo scheduling technique!

The additional time required for execution using the presented acyclic DDG modulo scheduling method as compared to the cyclic DDG modulo scheduling method is due to the additional code segments as discussed. **However, the major benefit of using the technique is still claimed to be the scalable performance benefit obtained from using an acyclic DDG modulo scheduling method.** To illustrate this benefit, the execution time calculations were performed with other resource availability conditions. The table in Figure 69 clearly illustrates the difference in performance of the techniques as more resources were made available.

available resources / scheduling method	2 adders 1 multiplier 1 branch 1 load/store	3 adders 1 multiplier 1 branch 1 load/store	5 adders 2 multipliers 1 branch 2 load/store	9 adders 3 multipliers 1 branch 3 load/stores
Cyclic DDG Modulo Scheduling	5.01	5.01	5.01	5.01
Suggested Acyclic DDG Modulo Scheduling	5.45	3.42	2.42	1.43
Bound On Performance	4.01	3.01	2.01	1.01

Figure 69: Average Time Units/Iteration For Various Configurations

The table displays the average number of time units per iteration as the number of resources is increased, using both a cyclic modulo scheduling technique and the technique presented in this thesis. The ideal solution provides the values which are the bound on best performance, being the lower bound on the average time per iteration.

For constructing the table, the scheduling procedure presented in the thesis was conducted for each of the resource availabilities shown. In all resource combinations tried, the compacted loop bound computation code required twelve time units. When the resources available were changed to three adders, one multiplier, one load/store unit, and one branch unit, N_{alive} became four, requiring compacted code segments for two iterations and one iteration. Compacted schedules were derived which consisted of twelve time units and ten time units, respectively, for these code segments.

When the resources available were again changed to five adders, two multipliers, two load/store units, and one branch unit, N_{alive} became five, requiring compacted code segments for four iterations, two iterations, and one iteration. Compacted schedules were derived which consisted of twelve time units, ten time units, and nine time units, respectively, for these code segments.

When the resources available were finally increased nine adders, three multipliers, three load/store units, and one branch unit, N_{alive} became nine, requiring compacted code segments for eight iterations, four iterations, two iterations, and one iteration. Compacted schedules were derived which consisted of fifteen time units, twelve time units, ten time units, and nine time units, respectively, for these code segments.

4. Comparison Of Techniques

As stated previously, the addition of the resources tends to reduce the lower bound on the IIII due to the resources constraints, and thereby increase in the bound on the performance. This is illustrated by the last row in Figure 69, which shows that when resources are added, the bound on the average iteration time units sequentially decreases with the bound on the IIII due to resources from four, to three, to two, to one time units.

However, if the IIII is also constrained by cyclic dependences, as when using the cyclic DDG modulo scheduling methods for scheduling, the IIII cannot be reduced below the limit imposed by the dependences no matter how many resources are made available. In the example, the original innermost loop code required a IIII due to dependence constraints of five time units. As more resources were made available, the IIII due to the dependence constraints remained at five time units. As a result, there was no effect on the cyclic DDG modulo scheduling performance, and the average time per iteration of the loop structure remained at 5.01 time units.

On the other hand, the loop pipelining method suggested in this paper is clearly affected by the availability of resources. For the example, adding resources resulted in the decrease of the IIII from five time units to one time unit. Because the IIII directly affects the performance, **the performance is also scalable with additional resources. This is the greatest advantage of using a method that utilizes acyclic DDG modulo scheduling.** As can be seen in Figure 69, the average iteration time decreased from 5.45 time units to 1.43 time units as more resources made available.

This demonstrates the significance of using an acyclic DDG modulo scheduling technique vice a cyclic DDG modulo scheduling technique. **Even with the additional overhead required for supporting the loop transformation to allow the acyclic DDG modulo scheduling, the performance of the proposed technique can exceed that of previously proposed cyclic DDG modulo scheduling techniques due to the scalable characteristic.**

It is recognized that the example only provides an illustration of the point being made, and is not a proof. **The actual performance of the technique presented is very much case dependent.** The performance of the technique is limited by the overhead require to compute the innermost loop bounds, to complete partial kernel execution in the epilog, and by the execution of non-pipelined iterations. While the computation of the innermost loop bounds may is fairly static, the value of N_{alive} influences the other two factors. As a result, the performance benefit of using the loop pipelining method proposed vice a cyclic DDG method may need to be determined on a case by cases basis. As a guideline, however, it is logical to believe that whichever method yields the lowest IIII, that method should be used.

This guideline seems even more reasonable when executing a large number of iterations. As the number of iterations of the innermost loop which are to be executed increases, the number of iterations executed by the pipelined loop section of code is also expected to increase. Consequently, the execution time of the entire structure becomes more and more dominated by the execution time of the pipelined iterations, and the average execution time per iteration approaches the IIII for the pipelining method used.

For the example, tables describing the average time units per iteration when the original upper bounds on the loop variables (N_1 and N_2) are altered are shown in Figure 70.

Figure 70.a is the same table shown in Figure 69. Figure 70.b and Figure 70.c show the results when the loop variable upper bounds were changed from the original example. In both cases, there is a drop in the average time per iteration when using the loop pipelining method presented from the original loop variable boundaries. This supports the expectation that the larger the number of iterations, the greater the percentage of iterations that use the pipelined kernel schedule and, hence, the closer the overall performance will approach the bound on performance.

It is important to note to that the performance gained from increasing the upper bound N_1 (as in Figure 70.b) is greater than that gained by increasing the upper bound N_2 (as in Figure 70.c). This too is expected, because increasing the value of N_1 for the example results in a "wider" transformed iteration space. That is, the transformation causes N_1 to be the new upper bound on the innermost loop control variable, and, consequently, increasing the value of N_1 results in a greater number of iterations to be executed within the innermost loop with the pipelined kernel schedule. Although increasing the value of N_2 add to the total number of iterations, it does not affect the "width" of the transformed iteration space, but rather the "length" (i.e., it adds to the number of innermost loop sequences executed). Although this results in a larger percentage of iteration being performed with the pipelined kernel, the additional overhead is required for executing each added innermost loop, which mitigates the performance gained.

5. Additional Improvements To Performance

As stated before, the performance of the technique is limited by the overhead require to compute the innermost loop bounds, to complete partial kernel execution in the epilog, and by the execution of non-pipelined iterations. Any improvement to the technique relies on elimination of unnecessary overhead in these areas.

available resources / scheduling method	2 adders 1 multiplier 1 branch 1 load/store	3 adders 1 multiplier 1 branch 1 load/store	5 adders 2 multipliers 1 branch 2 load/store	9 adders 3 multipliers 1 branch 3 load/stores
Cyclic DDG Modulo Scheduling	5.01	5.01	5.01	5.01
Suggested Acyclic DDG Modulo Scheduling	5.45	3.42	2.42	1.43
Bound On Performance	4.01	3.01	2.01	1.01

a. Average Time Units/Iteration With Original Loop Bounds of N_1=100 and N_2=500

available resources / scheduling method	2 adders 1 multiplier 1 branch 1 load/store	3 adders 1 multiplier 1 branch 1 load/store	5 adders 2 multipliers 1 branch 2 load/store	9 adders 3 multipliers 1 branch 3 load/stores
Cyclic DDG Modulo Scheduling	5.01	5.01	5.01	5.01
Suggested Acyclic DDG Modulo Scheduling	5.25	3.28	2.27	1.12
Bound On Performance	4.01	3.01	2.01	1.01

b. Average Time Units/Iteration With Loop Bounds of N_1=200 and N_2=500

available resources / scheduling method	2 adders 1 multiplier 1 branch 1 load/store	3 adders 1 multiplier 1 branch 1 load/store	5 adders 2 multipliers 1 branch 2 load/store	9 adders 3 multipliers 1 branch 3 load/stores
Cyclic DDG Modulo Scheduling	5.01	5.01	5.01	5.01
Suggested Acyclic DDG Modulo Scheduling	5.36	3.37	2.36	1.37
Bound On Performance	4.01	3.01	2.01	1.01

c. Average Time Units/Iteration With Loop Bounds of N_1=100 and N_2=1000

Figure 70: Average Time Units/Iteration With Various Loop Bound Values

Use of an efficient compaction routine is one way to ensure that the compacted code takes the minimum amount of time. Additional evaluation of the final code product may also be performed to identify where additional execution time can be saved.

The value of N_{alive} directly influences the length of the prolog/epilog as well as the number of non-pipelined iterations that must be performed. The reduction of N_{alive} when creating the pipelined schedule is also a method which can be used to help eliminate overhead. This was mentioned in Section III.B when discussing the scheduling algorithm to be used in creating the Modulo Resource Reservation Table.

B. ANALYSIS OF THE CODE GENERATION PROCEDURE

Analysis of the code generation procedure actually requires consideration of all the steps in the process for creating a new code loop from the original code loop. The steps that must be considered are the original transformation to create the final DDG, the modulo scheduling process, the code compaction procedures used in the code generation procedure, and the code generation procedure itself. Each of these is addressed below.

1. Complexity Of The Transformation

The transformation requires the determination of the value of the *sf* and the modification of the DDG to support the scheduling process.

To determine the *sf*, a depth first search can be done through the original DDG, with an evaluation done at each edge for use in the *sf* calculation. Assuming the Original DDG has V vertices and E edges, Tarjan [Ref. 15] describes how the determination can be done with complexity of order O(V+E).

The modification to the DDG requires the addition of as total of four nodes for the transformation and for the inclusion of the loop control instruction. For each of these four nodes, all other nodes in the DDG should be checked for dependence and a dependence arc created if need be. This operation has complexity of O(V). As a result, the overall complexity of creating the final DDG is O(V+E).

2. Complexity Of The Modulo Scheduling Process

The modulo scheduling process consists of creation of the modulo resource reservation table and the renaming of the registers as required.

a. Creating The Modulo Resource Reservation Table

To analyze the procedure for creating the modulo resource reservation table, the algorithm outlined in Section III.B can be used.

The initial calculation of the IIII requires an input for each node. If a depth first search is ag ı done to visit each node, the complexity of this calculation could be O(V+E).

According to Tarjan [Ref. 15], the search to determine the height of each node and the topological sort of the nodes to determine scheduling order can also be done in O(V+E).

In general, the procedure for scheduling instructions with **potentially different resource delays** into a modulo resource reservation table is a *bin packing* problem, which is an NP-Complete problem. However, our original assumption, and that assumption on which the algorithm presented in Section III.B is based, is that all resource delays are one time unit. This assumption reduces the complexity to that of the algorithm to a polynomial level. The main body of the procedure consists of a loop which is performed once for each node in the DDG. Within this loop a single node is picked (from the top of the topologically sorted list). All parents of this node are checked to determine earliest starting time. This really requires checking each edge coming into the node from parents, there is a upper bound on these edges of O(E). Also within this loop the node is scheduled in the table, which at most requires the consideration of IIII different time slots. However, an upper bound limit can be established on the IIII by the number of nodes. As a result, the overall complexity of the main body is O(V*(E + V)).

The overall complexity of the creation of the reservation table is therefore $O(V^2+VE)$.

b. Register Renaming Procedure

For register renaming, each instruction which defines a register value is considered and compared to all other instructions in which this definition is used. The lifetime of a register definition is determined based on the relative positions and iteration index (in the reservation table) of the instruction which defines the register and the instructions which used it. At most, each instruction could be dependent upon all other, and the lifetime calculated by determining all of these dependences. Consequently, the resultant lifetime determination for each register definition could be $O(V^2)$.

c. Overall Complexity

The overall complexity of pipelined kernel creation is a combination of the above complexities, which is $O(V^2+VE)$.

3. Complexity Of The Code Compaction Procedures

The code compaction procedures are used internal to the code generation procedure. Compaction is performed on both the loop bound calculation code segments and the non-pipelined iteration code segments. As with the creation of the modulo resource reservation table, a code compaction process aimed at creating the shortest code segment is again a bin packing problem, and therefore NP-Complete. However, simple scheduling heuristics can be applied, such as scheduling a selected node at the earliest time possible, which reduces the complexity to a polynomial level. The compaction procedures are analyzed assuming such heuristics are applied.

a. Compaction Of The Loop Bound Calculations

The loop bound calculations requires the use of known DDG's with known numbers of nodes and edges. The nodes can be scheduled following a topological sort of the graph and each node can be selected and scheduled at the earliest time possible. Because all of the elements are known, the procedure can be of constant order.

b. Compaction Of The Non-Pipelined Iterations

The number of non-pipelined iterations which can be compacted together at any one time is at most N_{alive}. Assuming that the nodes are scheduled in a manner to minimize N_{alive}, the upper bound on N_{alive} is linearly related to the number of instructions in the DDG. Hence, the number of nodes which have to be scheduled is on the order of V^2. To compact the iterations, a topological sort can be made of the final DDG. Assuming that the head vertices are known, the sort visits each edge once to create the sorted list (there are order O(VE) edges). However, a depth first search can still be conducted to label the heights initially. The result is that the sort would take $O(V^2+VE)$ steps. The actual scheduling only takes $O(V^2)$ steps, so the overall procedure would take $O(V^2+VE)$.

4. Complexity Of The Code Generation Procedure

The code generation procedure is relatively simple. One loop requires steps to be conducted for each of (n-3) loops, where n is the dimension of the original loop structure. This loop provide a complexity of O(n). Compaction of the loop bound calculations is included, but as noted above, this is of O(constant) = O(1).

The compaction of the non-pipelined iterations is done within a subloop which is executed at most $\log(N_{alive})$ times. As a result, the order of this subloop is $O(\log(V)*(V^2+VE))$.

One additional loop is executed on order of O(log(V)) as well.

The overall complexity of the code generation procedure is therefore $O(\log(V)*(V^2+VE)+ n)$

5. Overall Complexity

The overall complexity of the technique takes into account all of inputs from the components. The result is the complexity of $O(\log(V)*(V^2+VE)+ n)$.

VI. AN ISSUE OF DATA LOCALITY

To this point, the possible negative effects of the loop pipelining technique have been limited to the additional overhead that the technique may require. This, however, ignores the extremely important and realistic concern of memory access time.

To ensure high performance, a fast memory is essential to minimize the amount of delay that memory access instructions provide. It is possible that a single level memory may be used, in which case, memory is accessed at the same speed for every memory reference. Because the design of a fast cache for a VLIW machine is difficult, this is the approach taken by many VLIW machine designers. However, as with any other machine, the large main memory systems are relatively slow to any faster, smaller memory subsystems which can be incorporated in the design. It will be assumed, therefore, that the VLIW machine on which the technique will be performed has a upper level memory subsystem, like a cache, for faster access to reused memory data.

A. DATA LOCALITY

By adding a smaller, faster memory subsystem, designers are presuming that the principle of locality will hold in the target programs. This is generally true for programs which execute in their normal sequence, following the programmer's thought processes of sequential access to data arrays. In particular, loop structures tend to use the loop control variables to step through data structures in a sequential manner. As a result, different references to any one element tend to take place in localized time periods (*temporal locality*), and data elements stored in one small area of memory tend to be accessed in a localized time period (*spatial locality*). This then allows the reuse of data which has already been transferred to the transferred to the cache, saving the long delays for main memory data transfer by benefitting from the faster cache. In general, the better the locality in the referencing sequence, the more time saved in memory access.

However, as part of the pipelining technique, the original iteration space was skewed and permuted. While the original execution sequence was along the direction of the original control variables, the final execution sequence is along a transformed set of control variables--that is, along the direction of the wavefront The difference can be seen from the diagram shown in Figure 71, which shows the direction of execution of the wavefront as compared to the original loop control variables.

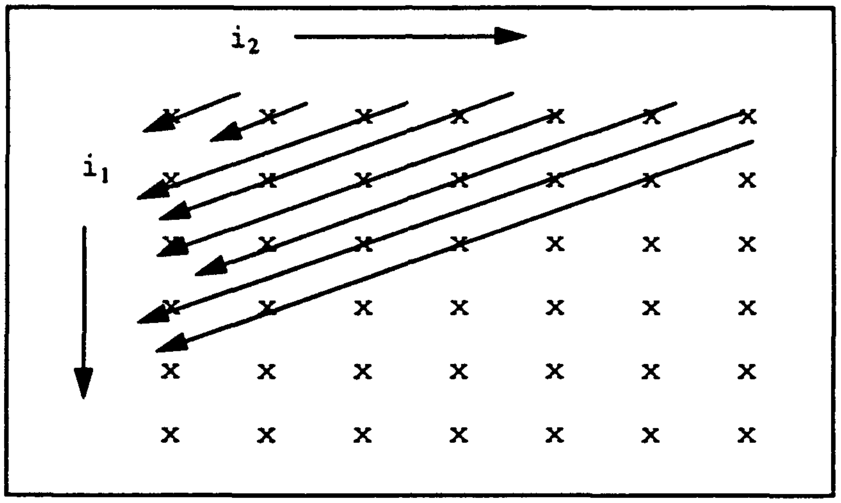


Figure 71: Wavefront Direction of Execution

The intent of the transformation was to eliminate data dependencies from the innermost loop iterations by ensuring that there are no data dependence along the innermost loop between consecutive iterations. Because data dependences are a subset of data reuses, at least some data reuses are eliminated from the innermost loop of the final loop structure. Consequently, executing the transformed loop structure along the innermost dimension does not benefit from the data locality from these data dependences that were present in the original loop structure

To illustrate the situation, consider Figure 72. This figure shows the original data dependence vectors of the iteration space originally presented in Figure 2 (Figure 72.a is the same as Figure 13 for the original example). One of the original dependences was between consecutive innermost iterations. After the transformation, the dependence is moved out of the innermost iterations to the outermost iterations.

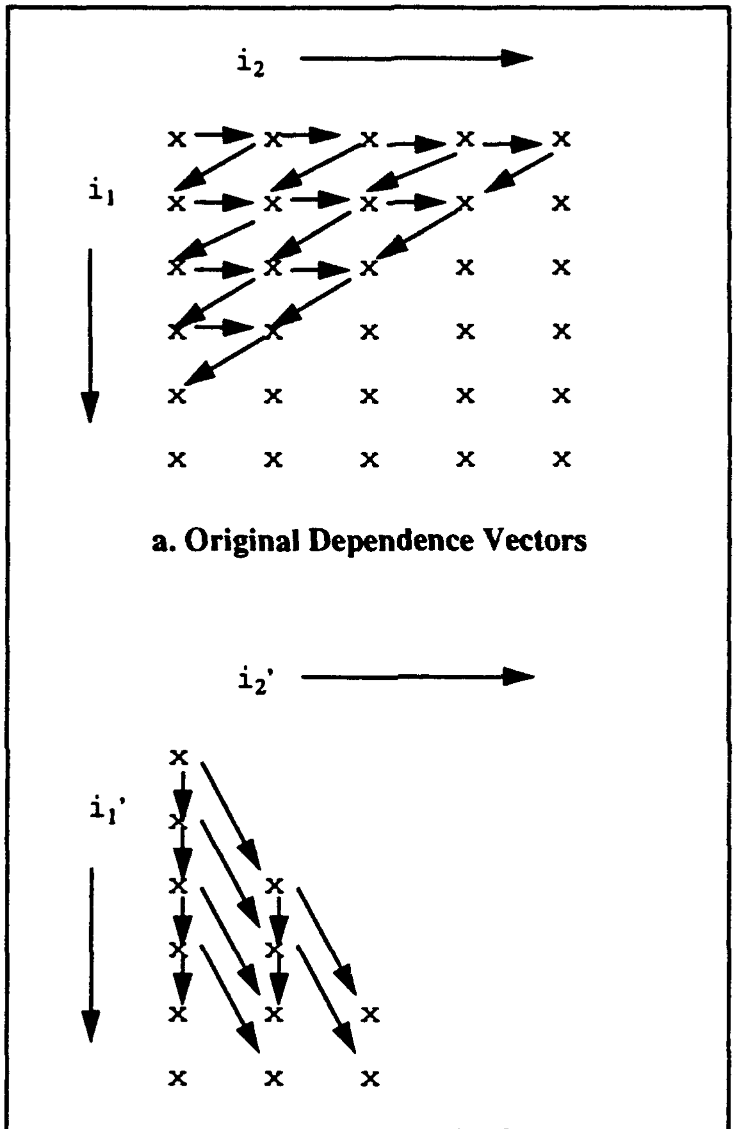

i_2
i_1
a. Original Dependence Vectors
i_2'
i_1'
b. Dependence Vectors in Transformed Iteration Space

Figure 72: Dependence Vector Alteration From Original To Transformed Iteration Space

If the cache size is smaller than the "row size" of the new iteration space, then it is probable that the data needed for an iteration from these dependences has been overwritten in the cache. For example, assume that the cache size was 64 words and each data element is one word long. In the original iteration space, a data value produced in a previous iteration is used by the current iteration, and should still be in the cache. However, in the transformed space, data produced in the previous second innermost iteration (i.e., previous row of the iteration space) will be used by the current iteration. Consequently, if more than 64 innermost loop iterations are being executed (i.e., if the row length is greater than 64 iterations), then the same data needed was produced at least 64 iterations in the past, and may have been overwritten during the wait. The chances of overwriting go up with the smaller cache dimension, obviously.

It is possible, therefore, that in an effort to eliminate data dependences between successive iterations of the innermost loop, the introduction of skewing and loop interchange is detrimental to the normal data locality of the loop structure. The actual effect, however, is very case dependent, relying on the value of loop bounds, array sizes, cache sizes, etc.

B. INVESITGATING THE DATA LOCALITY PROBLEM

In order to compare the effects of the pipelining technique loop transformation on reference locality, a program was written to create a reference trace of loop structure or a transformation from a loop structure. The trace generated from this program can be fed into the cache simulating tool DINEROIII[1], which computes statistics about cache misses and memory bus activity with various cache organizations and policies.

To investigate the effects of the loop transformation, reference traces were obtained for the original example loop structure from Figure 17, modified only in that the upper bound of the innermost and outermost loop variables were both set at 200 vice 100 and 500. The transformed loop generated will be the same as that generated in Section IV.C. The traces were performed assuming data size of one word each.

1. DINEROIII is a trace-driven cache simulating program that uses as input a sequence of memory references and outputs expected cache performance statistics. DINEROIII is authored by Mark D. Hill, Computer Science Department, University of Wisconsin, Madison, WI.

Because the testing was done to investigate the locality of data only, instruction fetches were not included in the reference listing. In all testing, the simulated cache was assumed to be a direct mapped cache, with a demand fetch policy, a write-back write policy, no sub-block access, and a write-allocate write policy. These choices were made somewhat arbitrarily, with the intent only to simplify the observations and maintain consistency Actual DineroIII statistical results of the testing is shown in the Appendix. The results depict a screen capture of the computer output of the statistics table that DINEROIII produces.

The initial evaluation was performed with a cache size of 128 words, with cache block size of one word. The most significant results are felt to be the percentage of references which resulted in misses and the total memory bus traffic. The results from this initial cache set-up shown in the table of Figure 73.

Block Size / Loop	One Word	Four Words
Sequential Loop Execution	67% 120200	17% 121208
Pipelined Loop Execution	66% 119623	48% 355464

percentage of miss rate

bus traffic in words

Figure 73: Miss Percentage and Total Bus Traffic with Each Loop Structure and With Cache Block Size of One or Four Words and Cache Size of 128 Words

The table compares the total data miss rate and bus traffic for both the original and the transformed loop structures. Each loop structure was also analyzed using a one word cache block and a four word cache block.

When the block size is one word, the percentage of misses and the total bus traffic for the original loop and the transformed loop are very close to being the same. This indicates that transformation of the loop structure did not adversely affect the temporal locality of the structure. However, when the block size was increased to four words, there was a dramatic difference between the two loop structures.

With the cache block size of four words, the miss rate of the original loop references dropped by a factor of about four from when the block size was one word. This was an expected result of the spatial locality that the original loop structure provided, given the sequential access of array elements along the innermost dimension.

As previously noted, the loop transformation performed to support loop pipelining the innermost loop eliminated the data dependences along the innermost dimension of the transformed structure, thereby eliminating of the spatial locality along this dimension. It was, therefore, expected as well that the greater block size would cause only a slight drop in miss rate from the one word block case. Additionally, because spatial locality is reduced, the total bus traffic would have to be much higher to support the data swapping of the four word blocks. This was supported by the data obtained.

Certainly, the specific results would change based on individual cases for cache configuration and loop code. To ensure that the results were not merely coincidental, the same analysis was performed with cache sizes of 512 words, 4k words, and 64 words. Figure 74 through Figure 76 show the results of the those tests.

Block Size / Loop	One Word	Four Words
Sequential Loop Execution	34% 80599	8% 121208
Pipelined Loop Execution	50% 100392	25% 220440

Figure 74: Miss Percentage and Total Bus Traffic with Each Loop Structure and With Cache Block Size of One or Four Words and Cache Size of 512 Words

Block Size / Loop	One Word	Four Words
Sequential Loop Execution	34% 80599	8% 81008
Pipelined Loop Execution	34% 80599	9% 81008

Figure 75: Miss Percentage and Total Bus Traffic with Each Loop Structure and With Cache Block Size of One or Four Words and Cache Size of 4k Words

Block Size / Loop	One Word	Four Words
Sequential Loop Execution	34% 80599	8% 81008
Pipelined Loop Execution	34% 80599	9% 81804

Figure 76: Miss Percentage and Total Bus Traffic with Each Loop Structure and With Cache Block Size of One or Four Words and Cache Size of 64k Words

When the cache size is small compared to the number of array elements (in the 128 word and 512 word cache), there is a significant difference in miss rate and bus traffic between the pipelined and non-pipelined loop structures when a four woιd block is used. This is attributed to spatial locality. As the cache size becomes large, the differences between the cache performances diminishes. This, however, is due to relative size of the cache to the data array.

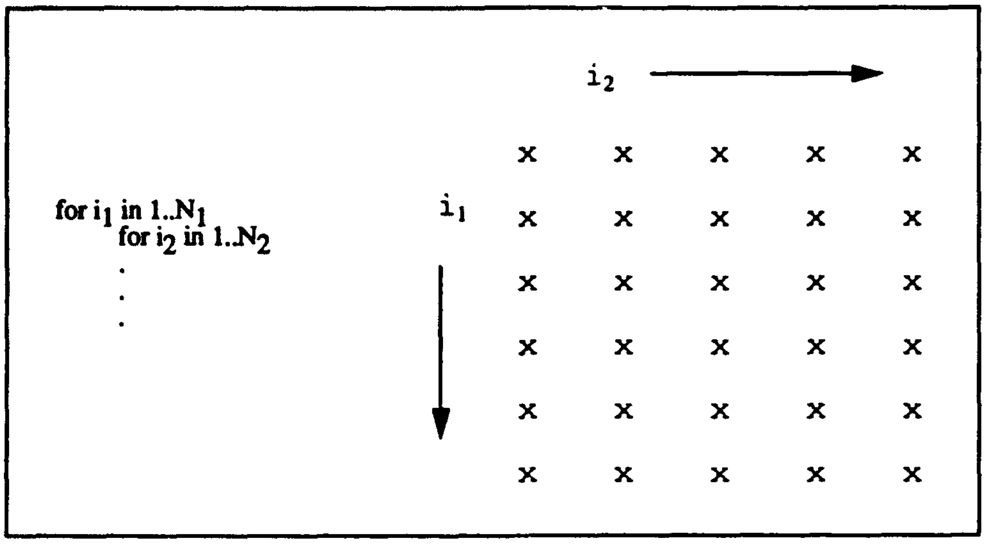
These few tests do suggest, however, that under certain conditions the pipelining method has the potential to disrupt the reference locality. This disruption would manifest itself during execution through a higher cache miss rate and bus traffic, resulting in an overall reduction of performance. In the worst case, the large delays caused by a greater number of main memory accesses may overshadow the performance gains of the pipelining technique.

C. A SOLUTION THROUGH TILING

One possible solution to reduce the cache misses is the application of iteration space tiling methods. Iteration space *Tiling* is a loop transformation method which partitions a loop structures normal iteration space into smaller sub-iteration spaces. For example, consider a loop structure and iteration space of Figure 77.

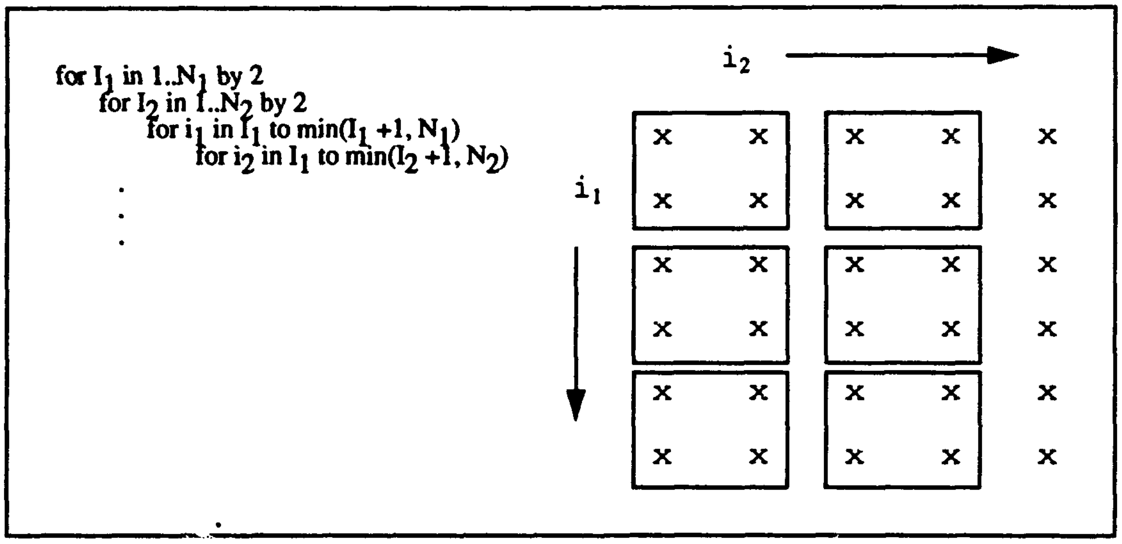
Tiling of this loop could result in a loop structure and a partitioned iteration space as in Figure 78. Each tiled section of the original iteration space is executed as a sub-space. The amount of data referenced in each of these tiles is a subset of the data referenced in the entire space. With this in mind, tiling has been presented as method by which data reuse can be optimized for a given cache size.

A description of how tiling can be applied to optimize the reuse of data is presented by Wolf and Lam [Ref. 16]. In using tiling to optimize the data reuse, an original loop structure is transformed into an equivalent loop structure for which the innermost nest containing some number of loops require a minimal number of memory accesses per iteration. This is



```
for $i_1$ in 1..$N_1$
    for $i_2$ in 1..$N_2$
        .
        .
        .
```

Figure 77: Untiled Loop Structure and Iteration Space



```
for $I_1$ in 1..$N_1$ by 2
    for $I_2$ in 1..$N_2$ by 2
        for $i_1$ in $I_1$ to min($I_1$ +1, $N_1$)
            for $i_2$ in $I_1$ to min($I_2$ +1, $N_2$)
    .
    .
    .
```

Figure 78: Tiled Loop Structure and Partitioned Iteration Space, with Tile Size of Two

done by performing unimodular transformations on the loop structure in an effort to make the all components of all data reuse vectors for the particular innermost nest of loops non-negative. Because data dependence vectors are a subset of the data reuse vectors, this has the additional effect of ensuring the innermost nest of loops becomes a fully permutable sub-loop structure.

By making the loops fully permutable, the innermost nest of loops can be interchanged in order to find the structure which best localizes the reuse of data, as well as providing the conditions for which the tiling transformation can be legally performed. The goal is to optimize cache data reuse by limiting the localized iteration space to a size for which all needed data can be contained within the cache at the same time. By selecting the a proper tile size, only a limited amount of data which is highly reused is placed in the cache at any one time.

The best selection of the tile size is one which uses a maximum of the cache (to improve data reuse), but avoids the mapping interference into cache locations. Demirhan [Ref. 17] provides an algorithm for choosing the best tile size based on cache size and data array row size in the case of directly mapped caches. In some instances, it is advisable to pad the array rows with empty elements in order to maximize the use of the cache.

1. Tiling With Loop Pipelining

The loop pipelining technique uses a transformation which eliminates data dependences along the innermost dimension. In performing this transformation, the two innermost loops were made fully permutable. Although the transformation was not motivated by any intent to tile, the result of the transformation is that the inner two dimensions of the final loop structure is in the same conditions required to allow tiling.

One major difference between the transformation done for the loop pipelining and that described by Wolf and Lam [Ref. 16] is that the loop pipelining technique did not perform skewing based on the data reuse vectors, but only on a subset of those vectors-- the data dependence vectors. However, because the transformation for loop pipelining has localized some reuse vectors within the two innermost loops, it is reasonable to believe that tiling may be able to recover some of the reference locality lost when making the innermost loop parallel. In particular, tiling a pipelined loop may still exploit the data reuse from the data dependence along the innermost dimension which was transferred to the second innermost loop dimension.

To investigate the possible application of tiling to the loop pipelined structure, the procedures given by Demirhan [Ref. 17] were conducted to identify the proper tile size for each of the cache configurations used in the last section[1]. Padded tiling was then applied to the original loop structure and transformed loop structures (the original loop structure required skewing in the innermost dimension as per Wolf and Lam [Ref. 16] to make it tilable). Reference traces were created for the tiled loop structures and tested with DINEROIII using the same cache configurations as previously chosen, except that in all cases, the cache block size was maintained at four words per block. The results of the testing is shown in Figure 79.

Cache Size / Loop	128 Words	512 Words	4k Words	64k Words
Sequential Loop Execution	12% 107140	10% 95020	9% 84112	9% 81804
Pipelined Loop Execution	9% 85680	9% 83620	9% 82396	9% 61952

Figure 79: Miss Rate and Total Bus Traffic with Padded Tiling Applied For Both the Original and Transformed Loop Structures

1. The optimal block size was determined using the algorithms provided in the reference. This included *padding* the array rows as necessary to maximally fill the cache. To accommodate for the skewing effect of the transformed loops, the actual row size needed was less by the total skewing of the loop structure, and was accounted for in the reference address calculations.

As can be seen from the table, the miss rate and the bus traffic have been greatly reduced for the pipelined loop compared to the non-tiled case. Additionally, both measures are for the transformed loop are at least as good as for the tiled original loop. The overall improvement is most noticeable with a smaller relative cache size, and diminishes as the cache size becomes so large as to be less effected by locality issues for the specific loop example.

In general, the results suggest that tiling might be used not only to reclaim some reference locality lost by the pipelining transformation, but also optimize the data reuse in the pipelined loop in the same manner as it can be used in other nested loop structures.

2. Potential Problems With Tiling

Although tiling may provide a dramatic improvement in reference locality, the application of tiling does not exist without a cost.

Even as prescribed by Wolf and Lam [Ref. 16], tiling generally requires loop transformations to provide the loop structure with a fully permutable loop. This requires the obvious overhead for loop transformation equations and code alteration. This overhead, however, is required of the loop pipelining technique anyway, and is therefore of no additional cost.

On the other hand, much of the overhead that the loop pipelining technique requires is that due to the transitioning into the pipelined schedule. This includes the sections of code for executing the prolog and the epilog as well as computation of the tile boundaries. By tiling, the iteration space is cut into smaller pieces, creating more boundaries. The result, it appears, would be a greater proportion of code dedicated to transition into and out of the pipelined segments, as well as more iterations performed in less efficient code segments (i.e., in the prolog and epilog). Roughly speaking, the overhead from a non-tiled to a tiled execution of a pipelined loop increases by a multiplicative factor of $(\frac{N_{n-1}}{square\ tile\ size})$.

The addition of this overhead is a major drawback to the use of tiling. The benefits of tiling must be weighed carefully against the cost of the overhead before it can be determined to be a feasible option. This is certainly an issue which needs further study.

D. THE EFFECT OF MULTIPLE LOAD/STORE UNITS

Thus far, the examples used for the observations about reference locality considered a target VLIW machine with only a single load/store unit. It is of obvious benefit to have a machine that used multiple load/store units to be able to concurrently access the memory for each of the units. With multiple load/store units, multiple references can be attempted for the same long instruction word. This will not only save time when the concurrent data accesses result in cache hits, but also when multiple concurrent data accesses result in misses. Consequently, the use of multiple load/store units should aid in reducing the penalty of the miss rate.

1. Investigating Concurrent Miss Savings

In an attempt to investigate the claim that multiple load/store units might result in reducing the miss penalty, again the example originating from Figure 17 was used. Following the loop pipelining technique presented in this thesis, pipelined schedules were generated assuming two load store units available and assuming three load store units available[1]. In the event that multiple load/store units are available, it is reasonable to implement a cache that has associativity which eliminates the possibility of self-interference within the same instruction. By setting the set associativity to at least the number of load/store units, same instruction interference is eliminated.

For the two pipelined schedule created, reference traces were generated assuming no tiling as well as assuming tiling, with two tile sizes being selected. Because there is no standard for choosing a specific tile size for the pipelined loop structures, the tile sizes that were chosen are somewhat arbitrary. However, to ensure consistency between the

1. The pipelined schedules were generated using the same configurations as used in the examples in Section V.A.3 when the multiple load store units were evaluated.

tests of various cache configurations the first size was selected using the optimization algorithm given by Demirhan [Ref. 17]. As stated before, this algorithm is intended to be used for directly mapped caches, so it serves no specific purpose in this case than to establish a standard method for picking the cache size. The second cache size was chosen to be the closest integer square root to the cache size. This also allows the choice is to be derived from a definite procedure which is consistent between the cache configurations.

The reference traces were again analyzed with DINEROIII. The cache configuration was set to simulate a set associative cache with the appropriate associativity for the number of load/store units available, a demand fetch policy, a write-back write policy, no sub-block access, and a write-allocate write policy. The cache block size was maintained at four words per block, to ensure the problems with spatial locality would be exhibited if they existed.

Because the intent of the test was to observe if the multiple load/store units could result in reducing the miss penalty, the output of DINERO was analyzed for reference misses which occurred in the same VLIW instruction. Because of the complexity of this analysis[1], the scope of the analysis was limited to examining only the references which occurred during the iterations which used the pipelined schedule, and ignored the areas of the iteration space that required sequential (or compacted) iteration execution.

Several tests were run with differing sizes of caches and differing number of load/store units. The results are summarized in Figure 80 through Figure 82. The smallest cache size examined was that with 512 words (Figure 80 and Figure 81). For both load/store unit configurations, the some savings in miss penalties were obtained when no tiling was used. When tiling was used, no miss penalty savings occurred.

1. DINEROIII analyzes reference traces assuming only sequential execution. The output, therefore, required analysis to determine which references occurred on which VLIW instruction lines. This is possible only for the iterations using the know pipelined schedule.

Although cache sizes up to 64k words were examined, results with all cache sizes above 512 words for both pipelined schedules indicated no saved miss penalty due to additional resources. Figure 82 is provided as a representative example of these results.

conditions	no tiling	padded tiling with tile size of 21	tiling with tile size of 22
total references	120,000	120,000	120,000
total pipelined references	115254	101054	101880
total pipelined instruction lines	76458	65894	66708
total pipelined misses	37000	8197	8208
total pipelined instruction lines with misses	36659	8197	8208
number of miss penalties saved	341	0	0
percent of miss penalties saved	1%	0	0

Figure 80: Summary of Investigation for Saving Miss Penalty With Two Load/Store Units, and a 512 Word, Two-Way Set Associative, Four Word Block Size Cache

conditions	no tiling	padded tiling with tile size of 21	tiling with tile size of 22
total references	120,000	120,000	120,000
total pipelined references	107742	72357	76248
total pipelined instruction lines	36448	26080	26856
total pipelined misses	37000	5412	5733
total pipelined instruction lines with misses	29913	5412	5733
miss penalties saved	7087	0	0
percent of miss penalty saved	19%	0	0

Figure 81: Summary of Investigation for Saving Miss Penalty With Three Load/Store Units, and a 512 Word, Four-Way Set Associative, Four Word Block Size Cache

conditions	no tiling	padded tiling with tile size of 41	tiling with tile size of 45
total references	120,000	120,000	120,000
total pipelined references	107796	92574	92574
total pipelined instruction lines	36448	31840	31840
total pipelined misses	8736	7276	7276
total pipelined instruction lines with misses	8736	7276	7276
miss penalties saved	0	0	0
percent of miss penalty saved	0	0	0

Figure 82: Summary of Investigation for Saving Miss Penalty With Three Load/Store Units, and a 2k Word, Four-Way Set Associative, Four Word Block Size Cache

2. Summary Of Results

The results obtained from the investigation of how multiple load/store units affect the miss penalty only illustrates the possibility that some penalty can be saved. Such savings, however, is dependent upon the specific loop structure, the number of load/store units, the pipelined schedule created, as well as the relative size of the cache as compared to the data array sized (or tile size). For the specific example observed, it appears that miss penalty is reduced slightly for those cases when the cache is relatively small and no tiling is used. The actual complex relationship between these factors is an area which requires additional study; however, if the results seen are at all representative, then the use of tiling may limit the miss rate savings with multiple load/store units.

E. SUMMARY OF DATA LOCALITY OBSERVATIONS

The observations made concerning the effects of the loop pipelining technique on data locality illustrates the complexity and case dependent nature of the problem. The results of the simple tests conducted indicate that data locality is negatively affected by the transformation process used to establish the conditions for the proposed loop pipelining technique. Particularly affected is the spatial locality that might normally exist in loop structures which use the loop control variables to regularly access data arrays.

The use of tiling transformations, however, appear promising in returning the level of locality of a pipelined loop to that of a non-pipelined loop, and virtually removing the negative effects of the transformation on data locality. Unfortunately, the benefit of the tiling must be weighed against the additional overhead that tiling imposes on the use of the pipelined code within each tile.

With multiple load/store units, some miss penalty might be saved if multiple misses occur within a single VLIW instruction. The conditions for which this occur, however, are very case dependent. In some instances, the uses of tiling may limit the ability for multiple instruction line misses to occur. The choice as to whether to use tiling, therefore, may also have to weigh the loss of savings in concurrent misses.

Although the observations made concerning the issue of data locality were limited, they identify the need for further study detailed study of the desired VLIW memory system and the effects of data locality optimization techniques used in conjunction with the loop pipelining technique presented in this thesis.

VII. CONCLUSION AND RECOMMENDATIONS

The technique for loop pipelining of perfectly-nested loop structures presented in this thesis combines previously well known methods of Wavefront Transformations and Acyclic DDG Modulo Scheduling to create a new loop pipelining method which is both simple and efficient. The resultant pipelined schedules produced are near-optimal for a given set of resources, with execution schedules varying from an ideal pipelining scenario only by the necessary addition of transformation instructions, boundary calculation overhead, and the addition of transition code necessary for use of the pipelined schedule.

Although the added overhead of the transformation tends to reduce performance, the technique is generally scalable with resource availability. This suggests that the addition of resources will improve performance beyond the limitations that present cyclic DDG modulo scheduling techniques face due to the bound that dependence cycles place on the final IIII. Because of this characteristic, the technique developed maintains a great advantage over previously proposed loop pipelining methods.

The code generation procedure described in Section IV.C.2 provides an extremely simple method to generate the final loop structure. The code generation procedure provides a systematic process by which to transform the original loop structure into a modified loop structure utilizing the loop pipelining technique presented. Most references tend to overlook this step when describing their techniques, but is an important and practical issue to address.

When developing the code generation procedure, code segments and their relationships were modelled with a DDG-type graph structure. This modeling proved extremely useful in providing a conceptual simplification and organization of the required code segments. The same modeling technique was used to describe the original loop structure, as well as used to develop the execution schedules presented for an ideal pipelining technique (see

Figure 67) and for a cyclic DDG Modulo Scheduling technique (see Figure 68). The ease at which the model was adaptable to other situations indicates that it might prove to be a valuable aid in future code restructuring investigations.

Observations indicate that spatial data locality may be adversely effected with the application of the presented loop pipelining technique. As a result, the use of cache memory systems when using the pipelining method could result in a higher cache miss rate. It is possible that the use of iteration space tiling techniques on the two innermost loops could overcome the negative effects, or that the existence of multiple load/store units may reduce the miss penalty with significant number of concurrent cache misses. However, the actual effect of the pipelining technique on data locality, the benefit of tiling, and the probability of concurrent cache misses appear case dependent on the original loop structure and on cache configuration. This must certainly be included in further study.

The work presented in this thesis is merely the beginning of a larger undertaking which must build upon and modify the current advancements. To obtain a clearer understanding of the performance benefits of the loop pipelining technique proposed, automated implementation of the method should be attempted. This would include the development of the data structures required for proper representation of DDGs, implementation of the loop transformation and modulo scheduling procedures, and implementation of the code generation procedure. Implementation of the code generation procedure will also require the implementation of code compaction sub-procedures. Once the loop pipelining method is automated, a greater number of examples can be examined, with simulated performance being evaluated to properly investigate the benefits of the pipelining method.

As mentioned previously, the issue of data locality should also be investigated further. Automation of the loop pipelining technique will also allow the examination of a greater number of examples to determine with more precision the effect that the pipelining technique has on data locality, as well as the saving multiple load/store units provide by allowing concurrent misses. Additionally, modifications to the code generation procedure can be made to investigate tiling affects. In particular, because tiling produces regularly sized blocks of iterations, it may be possible to simplify the boundary calculations or even overlap prolog and epilog executions within the tiles to gain efficiency. Associate with consideration for data locality is the issue of the appropriate choice of memory systems to best support VLIW machines in general, and loop pipelining in particular.

APPENDIX

This appendix contains the screen captured output of the program DINEROIII[1], displaying the results of simulated cache performance for reference traces from example code loops. The results were obtained in order to compare the effects of the pipelining technique loop transformation on reference locality. Traces from the original loop were compared with traces from a transformed loop specific configurations of cache size and block size. In all cases, the default settings of DINEROIII were used to maintain consistency. These default settings included simulation of a direct mapped cache with a demand fetch policy, and a write-back write policy. The only alterations which were allowed were in the cache size and the cache block size.

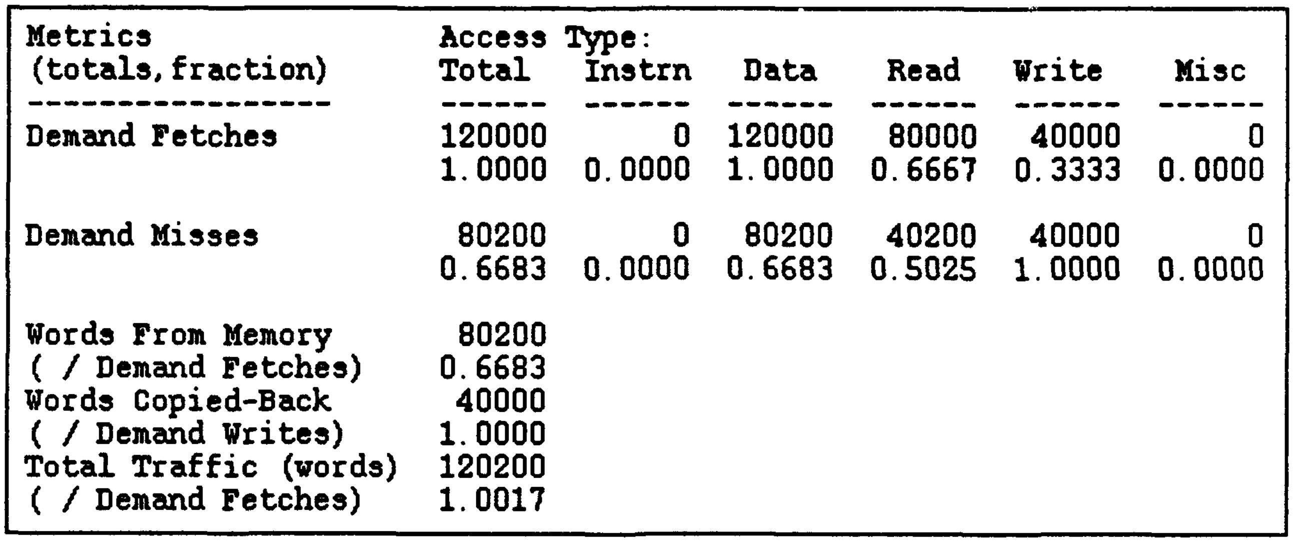
The tests were catagorized by the following list:

- cache size is primary division
- block size within cache, being either one word per block or four words per block. when block size was four words per block, the category is further divided as to whether the reference traces tested were obtained from tiled iteration spaces or non-tiled iteration spaces. When tiling, the tile size was chosen to be the largest tile size based on the cache size, with data array padding assumed to be applied as necessary per [Ref. 17] to avoid address interference
- for each category above, the test was performed on a reference trace from an original rectangular, non-pipelined loop structure, and then on a the references obtained from the transformed loop structure resulting from application of the loop pipelining technique described. In all cases, the original loop was a two dimensional loop structure, identical to the example shown in Figure 17, except that the upper limit for both the innermost and outermost loop variable was 200. The loop pipelining technique was performed as described in Section IV, with the final pipelined kernel schedule being the one produced in Figure 31 (specifically, only one load/store unit being available).

1. DINEROIII is a trace-driven cache simulating program that uses as input a sequence of memory references and outputs expected cache performance statistics. DINEROIII is authored by Mark D. Hill, Computer Science Department, University of Wisconsin, Madison, WI.

The screen capture diagrams which followed this categorization are given with explanitory captions in Figure 1 through Figure 24, divided by cache size.

A. TESTING WITH CACHE SIZE OF 128 WORDS

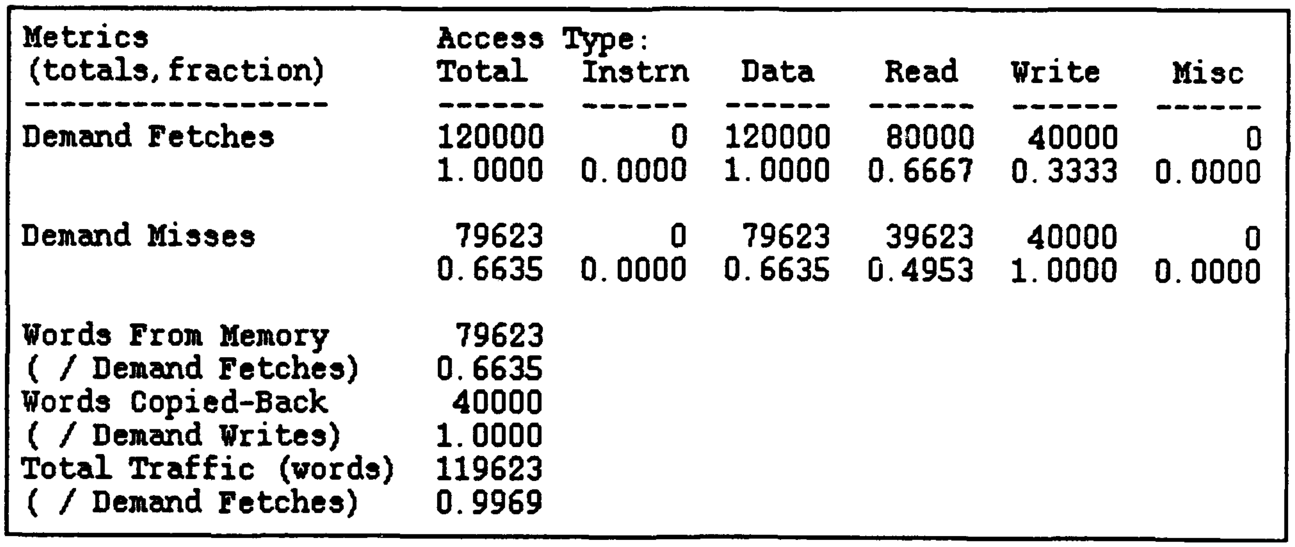


```
Metrics              Access Type:
(totals, fraction)   Total   Instrn    Data     Read    Write    Misc
-----------------    ------  ------   ------   ------   ------   ------
Demand Fetches       120000       0   120000    80000    40000        0
                     1.0000  0.0000   1.0000   0.6667   0.3333   0.0000

Demand Misses         80200       0    80200    40200    40000        0
                     0.6683  0.0000   0.6683   0.5025   1.0000   0.0000

Words From Memory     80200
( / Demand Fetches)  0.6683
Words Copied-Back     40000
( / Demand Writes)   1.0000
Total Traffic (words) 120200
( / Demand Fetches)  1.0017
```

Appendix Figure 1: Test Results For Reference Trace of Original Loop with Cache Size of 128 words, Cache Block Size of One Word, and No Tiling



```
Metrics              Access Type:
(totals, fraction)   Total   Instrn    Data     Read    Write    Misc
-----------------    ------  ------   ------   ------   ------   ------
Demand Fetches       120000       0   120000    80000    40000        0
                     1.0000  0.0000   1.0000   0.6667   0.3333   0.0000

Demand Misses         79623       0    79623    39623    40000        0
                     0.6635  0.0000   0.6635   0.4953   1.0000   0.0000

Words From Memory     79623
( / Demand Fetches)  0.6635
Words Copied-Back     40000
( / Demand Writes)   1.0000
Total Traffic (words) 119623
( / Demand Fetches)  0.9969
```

Appendix Figure 2: Test Results For Reference Trace of Pipelined Loop with Cache Size of 128 words, Cache Block Size of One Word, and No Tiling

```
Metrics               Access Type:
(totals, fraction)    Total    Instrn   Data     Read     Write    Misc
-----------------     ------   ------   ------   ------   ------   ------
Demand Fetches        120000        0   120000    80000    40000        0
                      1.0000   0.0000   1.0000   0.6667   0.3333   0.0000

Demand Misses          20201        0    20201    10201    10000        0
                      0.1683   0.0000   0.1683   0.1275   0.2500   0.0000

Words From Memory      80804
( / Demand Fetches)   0.6734
Words Copied-Back      40404
( / Demand Writes)    1.0101
Total Traffic (words) 121208
( / Demand Fetches)   1.0101
```

Appendix Figure 3: Test Results For Reference Trace of Original Loop with Cache Size of 128 words, Cache Block Size of Four Words, and No Tiling

```
Metrics               Access Type:
(totals, fraction)    Total    Instrn   Data     Read     Write    Misc
-----------------     ------   ------   ------   ------   ------   ------
Demand Fetches        120000        0   120000    80000    40000        0
                      1.0000   0.0000   1.0000   0.6667   0.3333   0.0000

Demand Misses          49586        0    49586    39586    10000        0
                      0.4132   0.0000   0.4132   0.4948   0.2500   0.0000

Words From Memory     198344
( / Demand Fetches)   1.6529
Words Copied-Back     157120
( / Demand Writes)    3.9280
Total Traffic (words) 355464
( / Demand Fetches)   2.9622
```

Appendix Figure 4: Test Results For Reference Trace of Pipelined Loop with Cache Size of 128 words, Cache Block Size of Four Words, and No Tiling

```
Metrics                  Access Type:
(totals, fraction)       Total    Instrn   Data     Read     Write    Misc
-----------------        ------   ------   ------   ------   ------   ------
Demand Fetches           120000        0   120000    80000    40000        0
                         1.0000   0.0000   1.0000   0.6667   0.3333   0.0000

Demand Misses             13885        0    13885     3885    10000        0
                         0.1157   0.0000   0.1157   0.0486   0.2500   0.0000

Words From Memory         55540
 ( / Demand Fetches)     0.4628
Words Copied-Back         51600
 ( / Demand Writes)      1.2900
Total Traffic (words)    107140
 ( / Demand Fetches)     0.8928
```

Appendix Figure 5: Test Results For Reference Trace of Original Loop with Cache Size of 128 words, Cache Block Size of Four Words, and Tiling

```
Metrics                  Access Type:
(totals, fraction)       Total    Instrn   Data     Read     Write    Misc
-----------------        ------   ------   ------   ------   ------   ------
Demand Fetches           120000        0   120000    80000    40000        0
                         1.0000   0.0000   1.0000   0.6667   0.3333   0.0000

Demand Misses             11220        0    11220     1220    10000        0
                         0.0935   0.0000   0.0935   0.0152   0.2500   0.0000

Words From Memory         44880
 ( / Demand Fetches)     0.3740
Words Copied-Back         40800
 ( / Demand Writes)      1.0200
Total Traffic (words)     85680
 ( / Demand Fetches)     0.7140
```

Appendix Figure 6: Test Results For Reference Trace of Pipelined Loop with Cache Size of 128 words, Cache Block Size of Four Words, and Tiling

B. TESTING WITH CACHE SIZE OF 512 WORDS

```
Metrics               Access Type:
(totals,fraction)     Total   Instrn   Data    Read    Write   Misc
-----------------     ------  ------  ------  ------  ------  ------
Demand Fetches        120000       0  120000   80000   40000       0
                      1.0000  0.0000  1.0000  0.6667  0.3333  0.0000

Demand Misses          40599       0   40599     599   40000       0
                      0.3383  0.0000  0.3383  0.0075  1.0000  0.0000

Words From Memory      40599
( / Demand Fetches)   0.3383
Words Copied-Back      40000
( / Demand Writes)    1.0000
Total Traffic (words)  80599
( / Demand Fetches)   0.6717
```

Appendix Figure 7: Test Results For Reference Trace of Original Loop with Cache Size of 512 words, Cache Block Size of One Word, and No Tiling

```
Metrics               Access Type:
(totals,fraction)     Total   Instrn   Data    Read    Write   Misc
-----------------     ------  ------  ------  ------  ------  ------
Demand Fetches        120000       0  120000   80000   40000       0
                      1.0000  0.0000  1.0000  0.6667  0.3333  0.0000

Demand Misses          60392       0   60392   20392   40000       0
                      0.5033  0.0000  0.5033  0.2549  1.0000  0.0000

Words From Memory      60392
( / Demand Fetches)   0.5033
Words Copied-Back      40000
( / Demand Writes)    1.0000
Total Traffic (words) 100392
( / Demand Fetches)   0.8366
```

Appendix Figure 8: Test Results For Reference Trace of Pipelined Loop with Cache Size of 512 words, Cache Block Size of One Word, and No Tiling

```
Metrics                    Access Type:
(totals, fraction)         Total   Instrn     Data     Read    Write     Misc
-----------------          ------  ------   ------   ------   ------   ------
Demand Fetches             120000       0   120000    80000    40000        0
                           1.0000  0.0000   1.0000   0.6667   0.3333   0.0000

Demand Misses               10151       0    10151      151    10000        0
                           0.0846  0.0000   0.0846   0.0019   0.2500   0.0000

Words From Memory           40604
 ( / Demand Fetches)       0.3384
Words Copied-Back           40404
 ( / Demand Writes)        1.0101
Total Traffic (words)       81008
 ( / Demand Fetches)       0.6751
```

Appendix Figure 9: Test Results For Reference Trace of Original Loop with Cache Size of 512 words, Cache Block Size of Four Words, and No Tiling

```
Metrics                    Access Type:
(totals, fraction)         Total   Instrn     Data     Read    Write     Misc
-----------------          ------  ------   ------   ------   ------   ------
Demand Fetches             120000       0   120000    80000    40000        0
                           1.0000  0.0000   1.0000   0.6667   0.3333   0.0000

Demand Misses               30235       0    30235    20235    10000        0
                           0.2520  0.0000   0.2520   0.2529   0.2500   0.0000

Words From Memory          120940
 ( / Demand Fetches)       1.0078
Words Copied-Back           99500
 ( / Demand Writes)        2.4875
Total Traffic (words)      220440
 ( / Demand Fetches)       1.8370
```

Appendix Figure 10: Test Results For Reference Trace of Pipelined Loop with Cache Size of 512 words, Cache Block Size of Four Words, and No Tiling

```
Metrics                 Access Type:
(totals,fraction)       Total   Instrn    Data    Read    Write    Misc
-----------------       ------  ------   ------  ------  ------  ------
Demand Fetches          120000       0   120000   80000   40000       0
                        1.0000  0.0000   1.0000  0.6667  0.3333  0.0000

Demand Misses            12147       0    12147    2147   10000       0
                        0.1012  0.0000   0.1012  0.0268  0.2500  0.0000

Words From Memory        48588
( / Demand Fetches)     0.4049
Words Copied-Back        46432
( / Demand Writes)      1.1608
Total Traffic (words)    95020
( / Demand Fetches)     0.7918
```

Appendix Figure 11: Test Results For Reference Trace of Original Loop with Cache Size of 512 words, Cache Block Size of Four Word, and Tiling

```
Metrics                 Access Type:
(totals,fraction)       Total   Instrn    Data    Read    Write    Misc
-----------------       ------  ------   ------  ------  ------  ------
Demand Fetches          120000       0   120000   80000   40000       0
                        1.0000  0.0000   1.0000  0.6667  0.3333  0.0000

Demand Misses            10755       0    10755     755   10000       0
                        0.0896  0.0000   0.0896  0.0094  0.2500  0.0000

Words From Memory        43020
( / Demand Fetches)     0.3585
Words Copied-Back        40600
( / Demand Writes)      1.0150
Total Traffic (words)    83620
( / Demand Fetches)     0.6968
```

Appendix Figure 12: Test Results For Reference Trace of Pipelined Loop with Cache Size of 512 words, Cache Block Size of Four Words, and Tiling

C. TESTING WITH CACHE SIZE OF 4096 WORDS

```
Metrics                 Access Type:
(totals, fraction)      Total   Instrn   Data     Read     Write    Misc
-----------------       ------  ------   ------   ------   ------   ------
Demand Fetches          120000       0   120000    80000    40000        0
                        1.0000  0.0000   1.0000   0.6667   0.3333   0.0000

Demand Misses            40599       0    40599      599    40000        0
                        0.3383  0.0000   0.3383   0.0075   1.0000   0.0000

Words From Memory        40599
( / Demand Fetches)     0.3383
Words Copied-Back        40000
( / Demand Writes)      1.0000
Total Traffic (words)    80599
( / Demand Fetches)     0.6717
```

Appendix Figure 13: Test Results For Reference Trace of Original Loop with Cache Size of 4096 words, Cache Block Size of One Word, and No Tiling

```
Metrics                 Access Type:
(totals, fraction)      Total   Instrn   Data     Read     Write    Misc
-----------------       ------  ------   ------   ------   ------   ------
Demand Fetches          120000       0   120000    80000    40000        0
                        1.0000  0.0000   1.0000   0.6667   0.3333   0.0000

Demand Misses            40599       0    40599      599    40000        0
                        0.3383  0.0000   0.3383   0.0075   1.0000   0.0000

Words From Memory        40599
( / Demand Fetches)     0.3383
Words Copied-Back        40000
( / Demand Writes)      1.0000
Total Traffic (words)    80599
( / Demand Fetches)     0.6717
```

Appendix Figure 14: Test Results For Reference Trace of Pipelined Loop with Cache Size of 4096 words, Cache Block Size of One Word, and No Tiling

```
Metrics               Access Type:
(totals, fraction)    Total   Instrn    Data    Read   Write    Misc
-----------------     ------  ------  ------  ------  ------  ------
Demand Fetches        120000       0  120000   80000   40000       0
                      1.0000  0.0000  1.0000  0.6667  0.3333  0.0000

Demand Misses          10151       0   10151     151   10000       0
                      0.0846  0.0000  0.0846  0.0019  0.2500  0.0000

Words From Memory      40604
( / Demand Fetches)   0.3384
Words Copied-Back      40404
( / Demand Writes)    1.0101
Total Traffic (words)  81008
( / Demand Fetches)   0.6751
```

Appendix Figure 15: Test Results For Reference Trace of Original Loop with Cache Size of 4096 words, Cache Block Size of Four Words, and No Tiling

```
Metrics               Access Type:
(totals, fraction)    Total   Instrn    Data    Read   Write    Misc
-----------------     ------  ------  ------  ------  ------  ------
Demand Fetches        120000       0  120000   80000   40000       0
                      1.0000  0.0000  1.0000  0.6667  0.3333  0.0000

Demand Misses          10251       0   10251     251   10000       0
                      0.0854  0.0000  0.0854  0.0031  0.2500  0.0000

Words From Memory      41004
( / Demand Fetches)   0.3417
Words Copied-Back      40800
( / Demand Writes)    1.0200
Total Traffic (words)  81804
( / Demand Fetches)   0.6817
```

Appendix Figure 16: Test Results For Reference Trace of Pipelined Loop with Cache Size of 4096 words, Cache Block Size of Four Words, and No Tiling

```
Metrics                 Access Type:
(totals,fraction)       Total   Instrn   Data     Read     Write    Misc
-----------------       ------  ------   ------   ------   ------   ------
Demand Fetches          120000       0   120000    80000    40000        0
                        1.0000  0.0000   1.0000   0.6667   0.3333   0.0000

Demand Misses            10611       0    10611      611    10000        0
                        0.0884  0.0000   0.0884   0.0076   0.2500   0.0000

Words From Memory        42444
( / Demand Fetches)     0.3537
Words Copied-Back        41668
( / Demand Writes)      1.0417
Total Traffic (words)    84112
( / Demand Fetches)     0.7009
```

Appendix Figure 17: Test Results For Reference Trace of Original Loop with Cache Size of 4096 words, Cache Block Size of Four Word, and Tiling

```
Metrics                 Access Type:
(totals,fraction)       Total   Instrn   Data     Read     Write    Misc
-----------------       ------  ------   ------   ------   ------   ------
Demand Fetches          120000       0   120000    80000    40000        0
                        1.0000  0.0000   1.0000   0.6667   0.3333   0.0000

Demand Misses            10399       0    10399      399    10000        0
                        0.0867  0.0000   0.0867   0.0050   0.2500   0.0000

Words From Memory        41596
( / Demand Fetches)     0.3466
Words Copied-Back        40800
( / Demand Writes)      1.0200
Total Traffic (words)    82396
( / Demand Fetches)     0.6866
```

Appendix Figure 18: Test Results For Reference Trace of Pipelined Loop with Cache Size of 4096 words, Cache Block Size of Four Word, and Tiling

D. TESTING WITH CACHE SIZE OF 64k WORDS

```
Metrics                 Access Type:
(totals, fraction)      Total   Instrn    Data    Read   Write    Misc
-----------------       ------  ------  ------  ------  ------  ------
Demand Fetches          120000       0  120000   80000   40000       0
                        1.0000  0.0000  1.0000  0.6667  0.3333  0.0000

Demand Misses            40599       0   40599     599   40000       0
                        0.3383  0.0000  0.3383  0.0075  1.0000  0.0000

Words From Memory        40599
( / Demand Fetches)     0.3383
Words Copied-Back        40000
( / Demand Writes)      1.0000
Total Traffic (words)    80599
( / Demand Fetches)     0.6717
```

Appendix Figure 19: Test Results For Reference Trace of Original Loop with Cache Size of 64k words, Cache Block Size of One Word, and No Tiling

```
Metrics                 Access Type:
(totals, fraction)      Total   Instrn    Data    Read   Write    Misc
-----------------       ------  ------  ------  ------  ------  ------
Demand Fetches          120000       0  120000   80000   40000       0
                        1.0000  0.0000  1.0000  0.6667  0.3333  0.0000

Demand Misses            40599       0   40599     599   40000       0
                        0.3383  0.0000  0.3383  0.0075  1.0000  0.0000

Words From Memory        40599
( / Demand Fetches)     0.3383
Words Copied-Back        40000
( / Demand Writes)      1.0000
Total Traffic (words)    80599
( / Demand Fetches)     0.6717
```

Appendix Figure 20: Test Results For Reference Trace of Pipelined Loop with Cache Size of 64k words, Cache Block Size of One Word, and No Tiling

```
Metrics                  Access Type:
(totals,fraction)        Total   Instrn    Data    Read   Write    Misc
-----------------        ------  ------  ------  ------  ------  ------
Demand Fetches           120000       0  120000   80000   40000       0
                         1.0000  0.0000  1.0000  0.6667  0.3333  0.0000

Demand Misses             10151       0   10151     151   10000       0
                         0.0846  0.0000  0.0846  0.0019  0.2500  0.0000

Words From Memory         40604
( / Demand Fetches)      0.3384
Words Copied-Back         40404
( / Demand Writes)       1.0101
Total Traffic (words)     81008
( / Demand Fetches)      0.6751
```

Appendix Figure 21: Test Results For Reference Trace of Original Loop with Cache Size of 64k words, Cache Block Size of Four Words, and No Tiling

```
Metrics                  Access Type:
(totals,fraction)        Total   Instrn    Data    Read   Write    Misc
-----------------        ------  ------  ------  ------  ------  ------
Demand Fetches           120000       0  120000   80000   40000       0
                         1.0000  0.0000  1.0000  0.6667  0.3333  0.0000

Demand Misses             10251       0   10251     251   10000       0
                         0.0854  0.0000  0.0854  0.0031  0.2500  0.0000

Words From Memory         41004
( / Demand Fetches)      0.3417
Words Copied-Back         40800
( / Demand Writes)       1.0200
Total Traffic (words)     81804
( / Demand Fetches)      0.6817
```

Appendix Figure 22: Test Results For Reference Trace of Pipelined Loop with Cache Size of 64k words, Cache Block Size of Four Words, and No Tiling

```
Metrics                  Access Type:
(totals, fraction)       Total   Instrn    Data    Read   Write    Misc
-----------------        ------  ------  ------  ------  ------  ------
Demand Fetches           120000       0  120000   80000   40000       0
                         1.0000  0.0000  1.0000  0.6667  0.3333  0.0000

Demand Misses             10251       0   10251     251   10000       0
                         0.0854  0.0000  0.0854  0.0031  0.2500  0.0000

Words From Memory         41004
( / Demand Fetches)      0.3417
Words Copied-Back         40800
( / Demand Writes)       1.0200
Total Traffic (words)     81804
( / Demand Fetches)      0.6817
```

Appendix Figure 23: Test Results For Reference Trace of Original Loop with Cache Size of 64k words, Cache Block Size of Four Words, and Tiling

```
Metrics                  Access Type:
(totals, fraction)       Total   Instrn    Data    Read   Write    Misc
-----------------        ------  ------  ------  ------  ------  ------
Demand Fetches            90371       0   90371   60248   30123       0
                         1.0000  0.0000  1.0000  0.6667  0.3333  0.0000

Demand Misses              7807       0    7807     276    7531       0
                         0.0864  0.0000  0.0864  0.0046  0.2500  0.0000

Words From Memory         31228
( / Demand Fetches)      0.3456
Words Copied-Back         30724
( / Demand Writes)       1.0200
Total Traffic (words)     61952
( / Demand Fetches)      0.6855
```

Appendix Figure 24: Test Results For Reference Trace of Pipelined Loop with Cache Size of 64k words, Cache Block Size of Four Words, and Tiling

LIST OF REFERENCES

1. Fisher, J., "Trace Scheduling: A Technique for Global Microcode Compaction", *IEEE Transactions on Computers*, Vol. C-30, No. 7, July 1981.

2. Lam, M. "Software Pipelining: An Effective Scheduling Technique for VLIW Machines", Conference on Programming Language Design and Implementation, Atlanta, Georgia, June 1988.

3. Zaky, A. and Sadayappan, P., "Optimal Static Scheduling of Sequential Loops on Multiprocessors", Proceedings of the International Conference on Parallel Processing, 1989.

4. Rau, B. and Glaeser, C., "Some Scheduling Techniques and an Easy Schedulable Horizontal Architecture for High Performance Scientific Computing", Proceedings of the Fourteenth Annual Workshop on Microprogramming, 1981.

5. Aiken, A. and Nicolau, A., "Perfect Pipelining: A New Loop Parallelization Technique", Department of Computer Sciences, Cornell University, 1987.

6. Rau, B., Schlansker, M. and Tirumalai, P., "Code Generation Schema for Modulo Scheduled Loops", Proceedings of the 25th International Symposium on Microarchitecture, 1992.

7. Zaky, A., "Efficient Static Scheduling of Loops on Synchronous Multiprocessors", Ph.D. Dissertation, Ohio State University, 1989.

8. Nicolau, A., "Loop Quantization: A Generalized Loop Unwinding Technique", *Journal of Parallel and Distributed Computing*, 1988.

9. Kim, K. and Nicolau, A., "N-Dimensional Perfect Pipelining"', Proceedings of the 25th Annual Hawaii International Conference on Systems Sciences, 1992.

10. Lamport, L., "The Parallel Execution of DO Loops", *Communications of the ACM*, February 1974.

11. Wolf, M. and Lam, M., "A Loop Transformation Theory and an Algorithm to Maximize Parallelism", *IEEE Transactions on Parallel and Distributed Systems*, July 1990.

12. Hsu, P., "Highly Concurrent Scalar Processing", Ph.D. Dissertation, University of Illinois, Urbana, 1986.

13. Hennessy, J., and Patterson, D., *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, 1990.

14. Colwell, R., Nix, R., O'Donnell, J., Papworth, D., and Rodman, P. "A VLIW Architeecture for a Trace Scheduling Compiler", *IEEE Transactions on Computers*, August 1988.

15. Tarjan, R. E. "Depth First Search And Linear Graph Algorithms", SIAM Journal on Computing, June 1972.

16. Wolf, M. and Lam, M., "A Data Locality Optimizing Algorithm", Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, June 1991.

17. Demirhan, A., "On Increasing The Effective Blocking Factor Of A Matrix For A Given Cache Organization", Master's Thesis, Naval Postgraduate School, Monterey, CA, 1992.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexanderia, VA 22304-6145	2
2.	Dudley Knox Library Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Dr. Ted Lewis Code 37, Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	1
4.	Dr. Amr M. Zaky Code CS/Za Associate Professor, Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	3
5.	Dr. Man-Tak Shing Code CS/Sh Associate Professor, Computer Science Department Naval Postgraduate School Monterey, CA 93943-5000	1
6.	Vicki H. Allen Department of Computer Science Utah State University Logan, Utah 84322-4205	1
7.	B. Ranakrishna Rau Cydrome, Inc. Milpitas, CA 95035	1
8.	Monica S. Lam Computer Systems Laboratory Stanford University Palo Alto, CA 94305	1
9.	LT Thor D. Aakre 136 Seal Ct. Marina, CA 93933	2